Архитектура проекта

“Корпоративный планировщик задач сотрудников”

Проект состоит из двух консольных приложений и одного с графическим интерфейсом пользователя.

1. “db\_creator” – Предназначено для создания базы данных сотрудников их задач и занесения в неё начальных значений.
2. “task\_server“ – Предназначено для непосредственной работы с базой данных и обработки сетевых запросов от подключенных клиентов.
3. “TaskClient” – Предоставляет графический интерфейс пользователя и возможность работы с базой данных, путём отправки запросов на сервер и обработки полученных ответов.

Далее более подробно о том, как устроено каждое приложение.

Приложение “db\_creator”.

Используемые библиотеки:

* Boost
* sqlite3

Приложение “db\_creator” принимает на вход один аргумент – имя файла для новой базы данных.

Часть библиотеки Boost “boost::filesystem” используем здесь только для того, чтобы узнать существует ли уже указанный при запуске программы файл, в котором будет храниться новая база данных. Если такой файл уже есть, в создании базы будет отказано.

Больше всего в этой программе используется библиотека “sqlite3”. С помощью её методов создаётся и заполняется база данных.

Для создания таблицы и заполнения её начальными данными используется набор заранее подготовленных sql запросов.

Алгоритм программы - следующий:

1. Создаётся база данных.
2. В базу добавляются 5 таблиц, перечисленные в файле “tables.xlsx”.
3. Таблицы “UserTypes” и “TaskStatuses” являются неизменяемыми и в них заносятся постоянные значения.
4. В оставшиеся три таблицы заносятся значения, позволяющие приступить к работе с базой, а именно три пользователя и несколько назначенных задач.

Для выполнения запросов используем лямбда функции и библиотечный метод sqlite3\_exec.

После выполнения всех sql запросов работа с базой прекращается, и программа завершает работу.

Приложение “task\_server”.

Используемые библиотеки:

* Boost
* sqlite3
* nlohmann\_json

Приложение “task\_server” принимает на вход два аргумента – номер порта для сетевых подключений и имя файла с базой данных.

Часть библиотеки Boost “boost::filesystem” используем чтобы проверить корректно ли указано имя файла с базой данных.

Часть библиотеки Boost “boost::asio” используем для асинхронной обработки сетевых подключений.

Библиотека sqlite3 используется для установки соединения и отправки запросов к базе данных.

Библиотека nlohmann\_json используется для формирования и обработки сетевых запросов, содержащих данные в формате JSON.

Формат JSON был выбран как наиболее понятный и распространённый формат для структурирования отправляемых данных.

*Классы, используемые в приложении* “task\_server”:

* TaskServer – Принимает сетевые подключения в асинхронном режиме. Знает какие команды/запросы могут приходить от подключенных клиентов. Создаёт сессии подключенных клиентов и хранит коллекцию созданных сессий. Для создания сессий использует класс ClientSession.
* DataStorage – открывает и контролирует доступ к базе данных. Реализует набор методов, которые, используя библиотеку sqlite3, позволяют получать данные из таблиц и изменять содержимое таблиц в базе.
* RequestManager – знает как устроена база данных – какие таблицы в ней есть. Предоставляет методы, позволяющие получать или изменять в базе данные определённого типа. Для доступа к базе использует класс DataStorage.
* ClientSession – представляет собой сессию, подключенного к серверу пользователя и знает тип данного пользователя. Принимает сетевые запросы от подключенного клиента и использует класс RequestManager для выполнения и предоставления того, что запрашивает клиент. Использует класс TaskServer для определения типа пришедшей команды/запроса, а также для инициализации своего завершения (т.е. для завершения текущей сессии).

*Инициализация работы приложения.*

В начале работы приложения создаётся “boost::asio::io\_context” и объект на основе класса TaskServer, после чего запускается цикл обработки событий при помощи метода run от io\_context.

Конструктор класса TaskServer создаёт объект на основе класса RequestManager, который в свою очередь, создаёт объект на основе класса DataStorage.

В конструкторе DataStorage инициализируется соединение с базой данных.

Конструктор класса TaskServer заполняет коллекции команд и запросов возможными типами и инициализирует асинхронный приём сетевых соединений.

С этого момента сервер находится в режиме ожидания, а первый объект на основе класса ClientSession будет создан при появлении нового подключения.

*Основной цикл работы сервера.*

При появлении нового подключения, сервер проверяет достигнуто ли максимальное кол-во одновременных подключений и, если это подтверждается, новое подключение будет отклонено.

Для каждого нового подключения создаётся свой сокет.

Сервер создаёт новую сессию пользователя, передавая в неё уже созданный сокет и уникальный идентификатор этой сессии. Также в сессию передаются указатель на сам TaskServer и на RequestManager.

Каждый раз, когда сервер создаёт очередную сессию пользователя, он добавляет её в коллекцию сессий, чтобы иметь возможность их корректно завершить в случае завершения своей работы.

После добавления в коллекцию, сервер запускает у сессии асинхронное чтение данных от подключенного клиента.

Когда всё сделано, сервер вновь переходит к ожиданию нового подключения.

*Основной принцип работы сессии пользователя.*

Как было написано выше, сессия пользователя находится в режиме асинхронного чтения данных из сокета, через который был подключен клиент.

При поступлении новых данных из сети, проверяется, что данные представлены в формате JSON.

Если данные получены в корректном формате, происходит обработка запроса клиента.

После обработки запроса, клиенту отправляется ответ в асинхронном режиме, а сессия возвращается к ожиданию/чтению новых данных из сети.

Для работы с данными в формате JSON сессия использует библиотеку nlohmann\_json, а для асинхронного приема и отправки данных по сети используется библиотека “boost::asio”.

*Подробнее об обработке запросов пользователя*.

Сначала определяется основной вид запроса

1. Тестовый запрос, предназначенный для проверки связи с сервером.
2. Запрос общедоступных данных, не требующих авторизации (логина) на сервере.
3. Логин (авторизация) пользователя на сервере.
4. Запрос на завершение работы сессии.
5. Запрос на завершение работы сервера.
6. Запрос на получение данных.
7. Запрос на добавление данных.
8. Запрос на изменение данных.
9. Запрос на удаление данных из базы.

Запрос на получение общих данных может быть двух видов:

1. Получение возможных типов пользователей и их описаний.
2. Получение возможных статусов задач и их описаний.

Виды запросов на получение данных:

1. Получить логин (username) и тип пользователя по уникальному id пользователя.
2. Получить ФИО пользователя по уникальному id пользователя.
3. Получить коллекцию из id всех пользователей.
4. Получить коллекцию из id всех задач.
5. По уникальному id задачи получить набор данных, связанных с этой задачей.

Виды запросов на добавление данных:

1. Добавить нового пользователя.
2. Добавить новую задачу.

Виды запросов на изменение данных:

1. Задать пользователю новый пароль.
2. Изменить тип пользователя.
3. Изменить статус задачи.
4. Назначить на задачу её исполнителя.
5. Изменить крайний срок выполнения задачи.

Виды запросов на удаление данных:

1. Удалить пользователя.
2. Удалить задачу.

Как можно заметить по представленным спискам, после определения основного вида запроса, для некоторых запросов определяется его подвид.

Класс ClientSession реализует полный набор методов для реагирования на каждый вид или, если нужно, подвид запросов от клиента. Для некоторых запросов реализована проверка уровня доступа подключенного пользователя. Например, удалить пользователя из базы может только пользователь типа “Администратор”, а если тип подключенного пользователя отличается, то в выполнении запроса будет отказано и клиенту будет отправлен ответ, содержащий соответствующую ошибку. Если проверка доступа прошла успешно и все необходимые для выполнения запроса поля JSON корректно заполнены, выполняется запрос к базе данных при помощи методов класса RequestManager, который, в свою очередь, будет использовать класс DataStorage для получения/изменения данных в базе.

Класс RequestManager знает расположение таблиц и данных в базе и используя данные, которые получены из запроса в рамках сессии пользователя, составляет часть тела для SQL запроса и передаёт его в соответствующий метод класса DataStorage.

Вызванный метод класса DataStorage формирует окончательный SQL запрос и пытается его выполнить, используя библиотеку sqlite3. Если запрос выполнить не удалось, сохраняется текст ошибки и предоставляется возможность его получить.

Т.к. к серверу может быть подключено несколько клиентов, работающих в асинхронном режиме, требуется защитить данные в базе от единовременного доступа. С этой целью, непосредственно перед выполнением запроса к базе, происходит попытка получить доступ к базе, данные которой защищены мьютексом. Сразу после выполнения запроса к базе, доступ к базе освобождается.

Основные виды ошибок, которые могут быть при обработке сетевого запроса от клиента:

1. Данные в запросе пришли не в формате JSON.
2. Получен пустой запрос.
3. Не удаётся представить данные в запросе как JSON объект.
4. Не удаётся определить основной тип запроса. Нет поля с типом запроса.
5. Тип запроса прочитан, но такой тип не известен серверу.
6. Для выполнения запроса в теле запроса отсутствует необходимый параметр.
7. Некорректное значение одного из параметров запроса.
8. Для выполнения такого запроса требуется авторизация пользователя, а она ещё не выполнена.
9. Попытка авторизации дважды в рамках одной сессии.
10. Недостаточный уровень доступа для выполнения указанного типа запроса.

Если оказывается, что запрос от клиента содержит ошибку, ему отправляется JSON ответ, содержащий следующие поля:

error : true  
 result : false  
 command : наименование запроса, на который отвечаем  
 errtext : здесь содержится описание ошибки в запросе

Значение true в поле error означает, что обнаружена проблема с самим запросом от клиента и дальнейшей попытки выполнить обращение к базе данных не было.

Могут быть и дополнительные поля:

parameter : название параметра, с которым что-то не так  
 details : описание, что именно не так с параметром

Если запрос был успешно обработан, то пользователю отправляется JSON ответ следующего вида:

error : false  
 command : наименование запроса, на который отвечаем  
 result : true / false

Значение false в поле error означает, что запрос от клиента был успешно принят и обработан.

Значение поля result зависит от того, удалось ли успешно выполнить обращение к базе данных и получить/изменить данные по запросу клиента.

В зависимости от запроса, в ответе могут быть и другие поля. Например, если был запрос типа “fullname” (узнать ФИО пользователя по id пользователя), то в случае успешного обращения к БД, ответ клиенту будет выглядеть как:

result : true  
second : фамилия  
first : имя  
patronymic : отчество

Если получить данные из базы не удалось, то ответ будет таким:

result : false  
details : “Unable to get FIO for provided user\_id”

Такое может быть, если переданный id пользователя отсутствует в базе данных.

*Краткое пояснение описанного выше о работе серверной части*:

* Сервер находится в постоянном ожидании новых подключений, но отвергает, если максимальное количество было достигнуто.
* Для принятого нового подключения создается новая сессия клиента, которая общается по сети через свой отдельный сокет.
* У сервера есть коллекция активных сессий для контроля над ними.
* Сервер умеет формировать SQL запросы к базе данных и защищать базу от единовременного доступа.
* Каждая активная сессия асинхронно читает и передаёт данные в сеть.
* Полученные от клиента запросы проверяются и обрабатываются в рамках отдельной сессии.
* Коммуникация между клиентом и сессией на сервере строится на формировании данных в JSON формате по определённым правилам.

Приложение “TaskClient”.

Используемые библиотеки:

* Boost
* Qt

Библиотека Qt используется для создания интерфейса пользователя и для работы с данными в формате JSON. Также она используется для предоставления данных в табличном виде.

Библиотека Boost используется для коммуникации по сети с сервером.

Для каждого типа пользователя реализовано окно с соответствующим интерфейсом, позволяющим проводить операции, доступные именно этому типу пользователя.

Клиентская часть, как и серверная, знает в каком JSON формате должны быть отправлены запросы и получены ответы.

Про назначение классов, реализованных в клиентской части можно прочитать в файле “project\_description.docx”.

Основная причина, по которой не использовал чужой, уже готовый протокол, это желание реализовать нечто своё, расширяемое, с использованием формата JSON, т.к. сейчас он довольно часто используется в клиент-серверных решениях, в частности, использующих REST (GET / POST запросы).

Если дальше совершенствовать проект, то думаю, имеет смысл добавить в клиентское приложение возможность нотификации пользователя о появлении новой задачи, назначенной на него или нотификации о приближении крайнего срока выполнения важной задачи.

Ещё одна доработка, которая напрашивается, это разделение и вывод задач по приоритетам выполнения.

Плюсом к выбору формата JSON можно указать возможность передачи XML документа, который мог бы содержать структурированное описание того, что нужно выполнить. Но тут нужно учитывать, что для удобочитаемости XML документ нуждается в html представлении, которое тоже можно передавать в рамках JSON.

Несомненно, вышеописанное можно было бы реализовать используя протокол CalDAV, но его изучение и внедрение займёт время, которого не так много)  
По сути было желание некий свой творческий подход к задаче реализовать.