|  |  |
| --- | --- |
| ДИСЦИПЛИНА | Технологии индустриального программирования |
| ИНСТИТУТ | ИПТИП |
| КАФЕДРА | Индустриального программирования |
| ВИД УЧЕБНОГО МАТЕРИАЛА | Методические указания к практическим занятиям по дисциплине |
| ПРЕПОДАВАТЕЛЬ | Александров Алексей Сергеевич |
| СЕМЕСТР | 3 семестр, 2024/2025 уч. год |

# Практическое занятие №12

**Сетевое взаимодействие между программами**

В настоящее время трудно представить себе такую программу, которая работала бы полностью в формате «оффлайн», т.е. вообще не требовала бы подключения к интернету для выполнения каких-либо функций.

Даже если весь функционал программы доступен без необходимости подключения к интернету, то, как минимум, для активации лицензии, необходимо единоразово подключиться к интернету.

Также, стоит обратить внимание, что многие программы или сервисы имею между собой интеграцию, например, для регистрации в сервисе пользователь может использовать аккаунт Яндекс, ВКонтакте и т.п.

Другой особенностью многих современных программ и сервисов является разделение функционала на несколько отдельных сервисов – «микросервисов» – суть заключается в разделении всего функционала системы на небольшие сервисы, взаимодействующие между собой, которые можно масштабировать при необходимости или добавлять, удалять или заменять.

Что объединяет все эти особенности? Все они используют интернет для взаимодействия – запрос на сервер для проверки лицензии, запрос на сервис Яндекса, ВКонтакте и т.п. для авторизации, запрос от одного микросервиса к другому и т.д. – все перечисленные методы выполняют запросы через интернет для получения или отправки информации. Но как они это делают?

Для понимания, как передаётся информация в интернете, необходимо обратиться к модели OSI:

Модель OSI (Open Systems Interconnection) — это эталонная модель, разработанная для описания функций телекоммуникационных или вычислительных систем, необходимых для сетевого взаимодействия. При этом сама по себе эталонная модель – не стандарт интернета, – её можно сравнить с фреймворками в языках программирования – в OSI «из коробки» доступны разные веб-стандарты — UDP, HTTP, FTP, Telnet и другие.

Сама модель OSI разделяет процесс сетевого взаимодействия на семь взаимосвязанных уровней, каждый из которых выполняет определённую функцию, например, передать данные или представить их в понятном для человека виде и т.п.

Представить модель OSI можно следующим способом:



Семислойная модель OSI

Рассмотрим уровни модели OSI и их функции:

1. Физический уровень (L1, physical layer). На самом нижнем уровне модели OSI данные представляют собой физические объекты — ток, свет или радиоволны. Они передаются по проводам или с помощью беспроводных сигналов.
2. Канальный уровень (L2, data link layer). Здесь данные должны преобразоваться в унифицированный формат, чтобы их можно было передавать на разные устройства и операционные системы.
3. Сетевой уровень (L3, Network Layer). На этом уровне происходит передача информации между различными локальными сетями с помощью таких устройств, как маршрутизатор.
4. Транспортный уровень (L4, transport layer). Обеспечивает надёжную передачу данных, контроль ошибок, сегментацию и повторную сборку данных. Гарантирует, что данные передаются без потерь и дубликатов.
5. Сеансовый уровень (L5, session layer). Устанавливает и завершает сеансы связи между устройствами, обеспечивает синхронизацию, управление длительностью соединения и восстановление после сбоев.
6. Уровень представления данных (L6, presentation layer). Преобразует информацию в стандартизированные форматы, чтобы гарантировать совместимость между отдельными системами.
7. Прикладной уровень (L7, application layer). Осуществляется взаимодействие пользователя с приложениями, обеспечивая веб-сёрфинг, электронную почту и передачу файлов.

На 7 уровне модели OSI, т.е. на прикладном уровне, реализуются протоколы HTTP, HTTPS и другие. Именно протокол HTTP и позволяет взаимодействовать, описанным выше, сервисам между собой.

Так, например, открывая веб-сайт, мы делаем HTTP запрос в браузере к определённому сайту. Например, отправляя запрос <http://www.mysite.com> мы видим страницу сайта, однако на самом деле, сервер в ответ на данный запрос отправляет HTML страницу, которую, наш браузер, в свою очередь, преобразует в понятное для человека представление сайта.

Однако точно такие же запросы могут выполнять, и приложения, и сайты друг к другу, точно также используя HTTP запросы, только вместо HTML они передают данные в формате XML и JSON – их не нужно показывать пользователю, поэтому они передаются в форматах, удобных для представления внутри программы.

Но зачем они это делают?

Простой ответ – это удобно! Нет необходимости писать огромный «монолит», когда можно, например, разделить систему на «фронтэнд» и «бэкенд» – по сути, на два отдельных приложения – одно написано на языке JavaScript, а второе на Java, C#, или Python, а возможно сразу на всех трёх языках – в формате микросервисов, которые передают данные между собой.

В данном случае фронтэнд отвечает за пользовательский интерфейс на клиентской стороне приложения. Это всё, что видит пользователь, когда открывает веб-страницу, и с чем он взаимодействует: кнопки, баннеры и анимация.

Бэкенд, в свою очередь, обеспечивает логические функции, нужные для работы сайта и приложения. Он работает на сервере и отправляет пользователю только результат работы.

Таким образом происходит чёткое деление функционала – фронтэнд выполняется на устройстве пользователя, а бэкенд на специальном сервере.

В таком случае взаимодействие происходит по принципу клиент-сервер.

Клиент — это устройство или программа, которая запрашивает услуги или данные у сервера. Клиент может быть веб-браузером, мобильным приложением или любой другой программой, которая взаимодействует с сервером.

Сервер — это мощное устройство или программа, которая предоставляет услуги или данные клиентам. Серверы могут обрабатывать множество запросов одновременно.

В данном случае, сторона фронтэнда является клиентом, а сторона бэкенда – сервером. Однако, возможна и ситуация, когда бэкенд становится клиентом, например, если необходимо выполнить авторизацию с помощью аккаунта Яндекс, в таком случае бэкенд становится клиентом, отправляющим запрос на сервер Яндекса.

Таким образом, клиент – это сторона, которая запрашивает данные, а сервер – сторона, которая эти данные отдаёт.

Однако, возникает вопрос, как сделать так, чтобы все сервисы «общались» единым, универсальным, способом, который можно было бы реализовать, используя любой язык программирования?

Для этого используют протокол HTTP и архитектурный стиль REST API.

**Принцип работы HTTP-запросов**

HTTP (HyperText Transfer Protocol) — это протокол прикладного уровня для передачи данных между клиентом и сервером через интернет. Он основан на модели клиент-сервер, где клиент отправляет запрос к серверу, а сервер отвечает на этот запрос.

Основные компоненты HTTP-запроса:

1. Метод — указывает, какое действие нужно совершить над ресурсом. Основные методы:

* GET — получение ресурса,
* POST — создание нового ресурса,
* PUT — обновление существующего ресурса,
* DELETE — удаление ресурса.

1. URL — адрес ресурса, к которому отправляется запрос.
2. Заголовки — дополнительные данные о запросе и ответе, такие как информация об авторизации, тип содержимого и др.
3. Тело — данные, которые отправляются вместе с запросом (например, при методах POST, PUT, PATCH). *В методе GET отсутствует тело запроса!*

Сервер обрабатывает запрос и возвращает ответ, который также состоит из нескольких частей:

1. Код состояния — числовой код, показывающий результат обработки запроса (например, 200 OK, 404 Not Found, 500 Internal Server Error).
2. Заголовки ответа.
3. Тело ответа — данные от сервера клиенту.

Здесь стоит особое внимание обратить на URL запроса (в любом методе). URL запроса формируется следующим образом:

<основной URL, или путь к ресурсу>**?**<параметры запроса>

Например:

http://www.site.ru/zapros?value1=data1

При этом, если параметров запроса несколько, они разделяются знаком «амперсанд» «&»:

http://www.site.ru/zapros?value1=data1&value2=data2

Таких параметров, соответственно, может быть любое количество.

Соответственно, знак амперсанда & используется для разделения нескольких параметров запроса, а каждый параметр состоит из пары «имя=значение», где имя и значение разделены знаком =. То есть:

* value1=data1 — это первый параметр,

где имя — value1, а значение — data1.

* value2=data2 — это второй параметр,

где имя — value2, а значение — data2.

Таким образом, строка запроса ?value1=data1&value2=data2 сообщает серверу, что клиент посылает два параметра запроса:

1. value1 со значением data1
2. value2 со значением data2

Сервер может использовать эти параметры для обработки запроса и возвращения соответствующих данных или выполнения определенных действий.

**Принцип работы REST API**

Перед тем, как перейти к REST API, необходимо определиться с понятием, что такое API в принципе?

API (Application Programming Interface) — это набор инструментов, которые позволяют взаимодействовать программам и приложениям друг с другом, использовать функции одного внутри другого.

Проще говоря, это набор правил, согласно которым общаются программы или модули в распределённых системах.

Примеры использования API:

* API мессенджеров Telegram и WhatsApp позволяют разработчикам программировать ботов.
* API Google Maps позволяет интегрировать карты Google в другие приложения и сайты, используя API-ключ.
* API «ВКонтакте» позволяет разработчикам создавать приложения, которые могут использовать информацию из профилей пользователей.
* API позволяет разработчику получать доступ к функциям одного приложения или веб-сайта из другого.

Методов реализации API много (Web, REST, SOAP, RPC, и др.), но самым популярным является REST API.

REST (Representational State Transfer) — архитектурный стиль для создания веб-сервисов, основанный на использовании HTTP-протокола. RESTful API — это интерфейс взаимодействия клиента и сервера, который следует принципам REST.

Основные принципы REST:

1. Ресурсно-ориентированный подход: все ресурсы идентифицируются уникальными URL-адресами.
2. Использование стандартных методов HTTP: GET, POST, PUT, DELETE.
3. Кэшируемость: ответы могут кэшироваться для улучшения производительности.
4. Отсутствие состояния: каждый запрос содержит всю необходимую информацию для выполнения операции, без сохранения состояния на стороне сервера.

Пример REST API:

GET /users/123 # Получение информации о пользователе с ID 123

POST /users # Создание нового пользователя

PUT /users/456 # Обновление информации о пользователе с ID 456

DELETE /users/789 # Удаление пользователя с ID 789

Соответственно, перед /users должен находиться URL сайта.

Таким образом, REST API устанавливает стандарт взаимодействия между различными программами, на основе протокола HTTP.

**Реализация REST API клиента в Qt**

Предположим, что у нас уже имеется сервер, написанный на одном из языков программирования, или предполагается интеграция с другим сервисом, предоставляющий доступ по REST API. Для этого рассмотрим пример создания клиента для REST API в Qt.

Qt предоставляет класс QNetworkAccessManager для выполнения сетевых операций, включая HTTP-запросы.

Для реализации REST API в Qt можно использовать следующие шаги:

Сначала необходимо подключить необходимые модули:

* Для системы сборки qmake:

В файле .pro проекта добавить модуль network:

QT += network

* Для системы сборки cmake:

find\_package(Qt6 REQUIRED COMPONENTS Network)

target\_link\_libraries(mytarget PRIVATE Qt6::Network)

Далее в заголовочном файле \*.h необходимо подключить класс QNetworkAccessManager:

#include <QNetworkAccessManager>

Также для выполнения запросов необходимо подключить классы QNetworkRequest, необходимый для выполнения запроса, и QNetworkReply – для получения ответа, соответвенно:

#include <QNetworkRequest>

#include <QNetworkReply>

После чего с помощью создания объекта QNetworkAccessManager можно выполнять HTTP запросы с помощью методов get(), post(), put() и deleteResource().

Стоит обратить внимание, что все запросы, выполняемые QNetworkAccessManager являются асинхронными, т.е. после отправки запроса, главный поток программы не останавливает своё выполнение для ожидания ответа, а продолжает своё выполнение. Поэтому, чтобы обработать ответ от сервера, необходимо, либо создать отдельный слот, обрабатывающий результат ответа, либо создать Event Loop, который будет ожидать получения ответа от сервера, тем самым останавливая выполнение программы, до получения ответа – такой способ позволяет выполнять синхронные вызовы.

Стоит учитывать, что использование синхронных вызовов упрощает логику кода программы, т.к. все вызовы по API выполняются последовательно, однако это замедляет выполнение параллельных вызовов.

Таким образом, создадим синхронный HTTP клиент, который будет позволять выполнять GET, POST, PUT и DELETE запросы.

Для начала создадим класс HttpClient и объявим методы для выполнения соответствующих запросов:



В качестве параметров методов указываем URL адрес запроса, JWT токен авторизации, с пустым значением по умолчанию, и указатель на строку для записи ошибок, в случае их возникновения. Также, для POST, PUT и DELETE запросов добавим JSON объект, который должен быть передан в запросе.

Для удобства работы с методами HTTP клиента, объявим их как static, чтобы доступ к ним был без создания объекта класса.

Для реализации GET запроса, в методе создания GET запроса создадим объекты QNetworkAccessManager, позволяющего выполнять запросы и QNetworkRequest – объект запроса.

QNetworkAccessManager manager;

QNetworkRequest request;

После чего запишем пустое значение в строку ошибки, если она передана, зададим URL запроса и установим заголовок токена авторизации:

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

Теперь, когда сформирован объект запроса, можно выполнить GET запрос:

// Выполняем GET запрос

QNetworkReply \*reply = manager.get(request);

Чтобы остановить выполнение программы, для ожидания выполнения запроса, создадим Event Loop с помощью класса QEventLoop, а также соединим слот finished() объекта ответа QNetworkReply на выполняемый запрос со слотом quit() созданного Event Loop, для завершения цикла при получении ответа от сервера:

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

Получив ответ, его можно обработать. Проверим, что запрос выполнился без ошибок, а если есть ошибки, то запишем полученную ошибку в строку для ошибок:

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

if(reply->error() == QNetworkReply::ContentAccessDenied)

{

\*errorString = "Доступ запрещён!";

}

else

{

\*errorString = reply->errorString();

}

}

После чего пропадает необходимость хранения объекта ответа, а значит, можно освободить занимаемую им память:

// Освобождаем ресурсы

reply->deleteLater();

А полученный ответ возвращаем как результат выполнения функции:

return QString::fromUtf8(response);

Таким образом полный код выполнения GET запрос выглядит следующим образом:

QString HttpClient::sendGetRequest(QUrl url, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Выполняем GET запрос

QNetworkReply \*reply = manager.get(request);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

if(reply->error() == QNetworkReply::ContentAccessDenied)

{

\*errorString = "Доступ запрещён!";

}

else

{

\*errorString = reply->errorString();

}

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

Аналогично реализуем метод выполнения POST запросов, заменив метод get() на post() объекта QNetworkAccessManager:

QString HttpClient::sendPostRequest(QUrl url, QJsonObject json, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Выполняем POST запрос

QNetworkReply \*reply = manager.post(request);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

Однако, стоит учитывать, что запросы POST, PUT и DELETE отправляют тело запроса в формате JSON, которого нет у GET запроса, поэтому, после добавления заголовка токена, добавим заголовок ContentType со значением application/json, а также сформируем JSON объект и добавим его как параметр метода post():

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Создаем JSON объект для передачи в теле запроса

QByteArray postData = QJsonDocument(json).toJson();

// Выполняем POST запрос

QNetworkReply \*reply = manager.post(request, postData);

Таким образом, полный код POST запроса будет выглядеть следующим образом:

QString HttpClient::sendPostRequest(QUrl url, QJsonObject json, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Создаем JSON объект для передачи в теле запроса

QByteArray postData = QJsonDocument(json).toJson();

// Выполняем POST запрос

QNetworkReply \*reply = manager.post(request, postData);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

Теперь осталось только реализовать методы PUT и DELETE – по сути они по своей реализации не отличаются от POST и GET запросов.

Для реализации метода PUT достаточно в методе для выполнения POST запроса заменить вызов метода post() на вызов метода put():

QString HttpClient::sendPutRequest(QUrl url, QJsonObject json, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Создаем JSON объект для передачи в теле запроса

QByteArray putData = QJsonDocument(json).toJson();

// Выполняем PUT запрос

QNetworkReply \*reply = manager.put(request, putData);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

А для реализации метода DELETE возьмём метод, реализующий GET запрос и заменим вызов метода get() на вызов метода deleteResource():

QString HttpClient::sendDeleteRequest(QUrl url, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Выполняем DELETE запрос

QNetworkReply \*reply = manager.deleteResource(request);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

Полный код класса HttpClient:

QString HttpClient::sendGetRequest(QUrl url, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Выполняем GET запрос

QNetworkReply \*reply = manager.get(request);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

if(reply->error() == QNetworkReply::ContentAccessDenied)

{

\*errorString = "Доступ запрещён!";

}

else if(reply->error() == QNetworkReply::ContentNotFoundError)

{

\*errorString = "Ошибка 404! Не найдено!";

}

else

{

\*errorString = reply->errorString();

}

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

QString HttpClient::sendPostRequest(QUrl url, QJsonObject json, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Создаем JSON объект для передачи в теле запроса

QByteArray postData = QJsonDocument(json).toJson();

// Выполняем POST запрос

QNetworkReply \*reply = manager.post(request, postData);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

QString HttpClient::sendPutRequest(QUrl url, QJsonObject json, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Создаем JSON объект для передачи в теле запроса

QByteArray putData = QJsonDocument(json).toJson();

// Выполняем PUT запрос

QNetworkReply \*reply = manager.put(request, putData);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

QString HttpClient::sendDeleteRequest(QUrl url, QString token, QString \*errorString)

{

QNetworkAccessManager manager;

QNetworkRequest request;

// Если передана переменная для записи ошибок

if(errorString != nullptr)

{

\*errorString = ""; // Записываем пустую строку, ошибок нет

}

// Устанавливаем URL для запроса

request.setUrl(url);

// Устанавливаем заголовок Authorization

if(!token.isEmpty())

{

QString authorizationHeader = "Bearer " + token;

request.setRawHeader("Authorization", authorizationHeader.toUtf8());

}

// Устанавливаем заголовок Content-Type

request.setHeader(QNetworkRequest::ContentTypeHeader, "application/json");

// Выполняем DELETE запрос

QNetworkReply \*reply = manager.deleteResource(request);

// Ожидаем, пока запрос завершится

QEventLoop loop;

QObject::connect(reply, SIGNAL(finished()), &loop, SLOT(quit()));

loop.exec();

// Читаем ответ

QByteArray response = reply->readAll();

// Обрабатываем ответ

if (reply->error() != QNetworkReply::NoError && errorString != nullptr)

{

\*errorString = reply->errorString();

}

// Освобождаем ресурсы

reply->deleteLater();

return QString::fromUtf8(response);

}

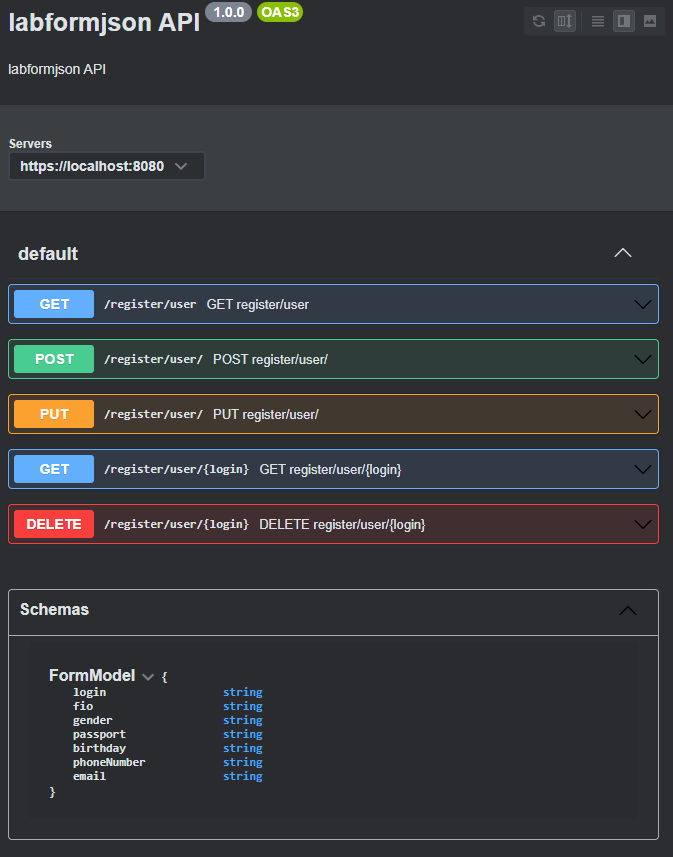
Таким образом был реализован класс HttpClient, позволяющий выполнять методы GET, POST, PUT и DELETE, для взаимодействия с внешними сервисами, с помощью REST API.

**Использование REST API клиента в Qt**

Для использования созданного HTTP клиента можно использовать REST API сервер, или написать его самостоятельно. Для демонстрации работы, мы создадим небольшое бэкенд-приложение на языке Java, со следующими функциями:

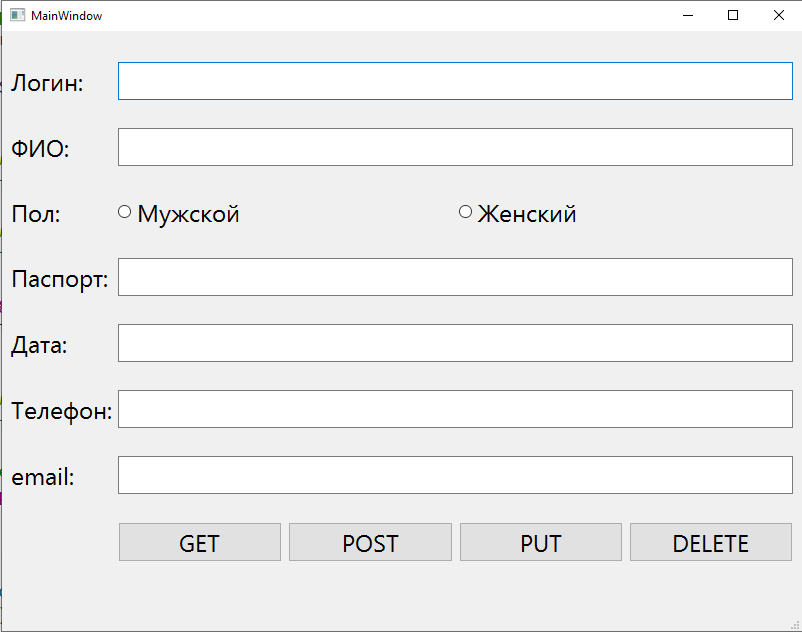
* GET /register – получить все данные о всех регистрациях
* GET /register/{login} – получить данные о регистрации по логину
* POST /register – создать новую запись о регистрации
* PUT /register/{login} – обновить данные регистрации по логину
* DELETE /register/{login} – удалить запись о регистрации по логину

Документация сервера на языке Java будет выглядеть следующим образом:



Теперь необходимо реализовать REST API клиент в Qt.

Для этого создадим следующую форму:



И реализуем обработку кнопок с использованием HttpClient:

void MainWindow::on\_pushButton\_get\_clicked()

{

QString error;

QString login = ui->lineEdit\_login->text();

QString response = HttpClient::sendGetRequest(

QUrl(HOST\_URL + "/register/user/" + login), "", &error);

if(error.isEmpty())

{

QMessageBox::information(this, "", response);

QJsonObject jsonObject = QJsonDocument::fromJson(response.toUtf8()).object();

QString login = jsonObject["login"].toString();

QString fio = jsonObject["fio"].toString();

QString date = jsonObject["birthday"].toString();

QString gender = jsonObject["gender"].toString();

QString passport = jsonObject["passport"].toString();

QString phoneNumber = jsonObject["phoneNumber"].toString();

QString email = jsonObject["email"].toString();

ui->lineEdit\_login->setText(login);

ui->lineEdit\_fio->setText(fio);;

ui->radioButton\_male->setChecked(gender == "male" ? true : false);

ui->radioButton\_female->setChecked(gender == "female" ? true : false);

ui->lineEdit\_date->setText(date);

ui->lineEdit\_passport->setText(passport);

ui->lineEdit\_phone->setText(phoneNumber);

ui->lineEdit\_email->setText(email);

}

else

{

QMessageBox::critical(this, "", error);

}

}

void MainWindow::on\_pushButton\_post\_clicked()

{

// считываем данные

QString login = ui->lineEdit\_login->text();

QString fio = ui->lineEdit\_fio->text();

QString gender = ui->radioButton\_male->isChecked() ? "male" : "female";

QString passport = ui->lineEdit\_passport->text();

QString date = ui->lineEdit\_date->text();

QString phoneNumber = ui->lineEdit\_phone->text();

QString email = ui->lineEdit\_email->text();

// формируем JSON

// Создание JSON-объекта

QJsonObject jsonObject;

jsonObject["login"] = login;

jsonObject["fio"] = fio;

jsonObject["gender"] = gender;

jsonObject["passport"] = passport;

jsonObject["birthday"] = date;

jsonObject["phoneNumber"] = phoneNumber;

jsonObject["email"] = email;

// Отправка запроса

QString error;

QString response = HttpClient::sendPostRequest(

QUrl(HOST\_URL+"/register/user/"), jsonObject, "", &error);

if(error.isEmpty())

{

QMessageBox::information(this, "", response);

}

else

{

QMessageBox::critical(this, "", error);

}

}

void MainWindow::on\_pushButton\_put\_clicked()

{

// считываем данные

QString login = ui->lineEdit\_login->text();

QString fio = ui->lineEdit\_fio->text();

QString gender = ui->radioButton\_male->isChecked() ? "male" : "female";

QString passport = ui->lineEdit\_passport->text();

QString date = ui->lineEdit\_date->text();

QString phoneNumber = ui->lineEdit\_phone->text();

QString email = ui->lineEdit\_email->text();

// формируем JSON

// Создание JSON-объекта

QJsonObject jsonObject;

jsonObject["login"] = login;

jsonObject["fio"] = fio;

jsonObject["gender"] = gender;

jsonObject["passport"] = passport;

jsonObject["birthday"] = date;

jsonObject["phoneNumber"] = phoneNumber;

jsonObject["email"] = email;

// Отправка запроса

QString error;

QString response = HttpClient::sendPutRequest(

QUrl(HOST\_URL+"/register/user/"), jsonObject, "", &error);

if(error.isEmpty())

{

QMessageBox::information(this, "", response);

}

else

{

QMessageBox::critical(this, "", error);

}

}

void MainWindow::on\_pushButton\_delete\_clicked()

{

QString error;

QString login = ui->lineEdit\_login->text();

QString response = HttpClient::sendDeleteRequest(

QUrl(HOST\_URL + "/register/user/" + login), "", &error);

if(error.isEmpty())

{

QMessageBox::information(this, "", response);

}

else

{

QMessageBox::critical(this, "", error);

}

}

В результате метод GET будет получать информацию с сервера по данному логин, метод POST создает запись на сервере с текущими данными, метод PUT обновляет данные на сервере по данному логину, а метод DELETE удаляет запись с данным логином.

**Источники информации**

1. Дмитрий Зверев Сетевая модель OSI: 7 уровней, их протоколы и функции — гайд для новичков / Skillbox Media / Дмитрий Зверев [Электронный ресурс] // Skillbox Media : [сайт]. — URL: <https://skillbox.ru/media/code/chto-takoe-model-osi-i-zachem-ona-nuzhna-prepariruem-sloyenyy-pirog-interneta/> (дата обращения: 17.11.2024).
2. Евгений Кучерявый API — что это такое: простыми словами об интерфейсах и интеграции по API / Skillbox Media / Евгений Кучерявый [Электронный ресурс] // Skillbox Media : [сайт]. — URL: <https://skillbox.ru/media/code/chto_takoe_api/> (дата обращения: 17.11.2024).
3. Редакция «Код» Skillbox Media Методы GET и POST HTTP-запросов: что делают и в чём разница между ними / Skillbox Media / Редакция «Код» Skillbox Media [Электронный ресурс] // Skillbox Media : [сайт]. — URL: <https://skillbox.ru/media/code/metody-get-i-post-v-http/?utm_source=media&utm_medium=link&utm_campaign=all_all_media_links_links_articles_all_all_skillbox> (дата обращения: 17.11.2024).
4. Иван Стуков REST API: что это такое, как работает и для чего используется / Skillbox Media / Иван Стуков [Электронный ресурс] // Skillbox Media : [сайт]. — URL: <https://skillbox.ru/media/code/rest-api-chto-eto-takoe-i-kak-rabotaet/?utm_source=media&utm_medium=link&utm_campaign=all_all_media_links_links_articles_all_all_skillbox> (дата обращения: 17.11.2024).
5. OTUS Journal HTTP-запросы от А до Я OTUS / OTUS Journal [Электронный ресурс] // OTUS : [сайт]. — URL: <https://otus.ru/journal/http-zaprosy-ot-a-do-ya/> (дата обращения: 17.11.2024).
6. Блог Яндекс Практикума Четыре буквы, на которых держится интернет: что такое протокол HTTP и почему на нём работают почти все сайты / Блог Яндекс Практикума [Электронный ресурс] // Блог Яндекс Практикума : [сайт]. — URL: <https://practicum.yandex.ru/blog/chto-takoe-protokol-http/> (дата обращения: 17.11.2024).
7. Qt QNetworkAccessManager Class | Qt Network 6.8.0 / Qt [Электронный ресурс] // Qt Documentation : [сайт]. — URL: <https://doc.qt.io/qt-6/qnetworkaccessmanager.html> (дата обращения: 17.11.2024).