МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
"Южно-Уральский государственный университет"
(национальный исследовательский университет)

Образовательная программа
"Технологии интернета вещей"
по направлению подготовки
09.04.01 «Информатика и вычислительная техника»
(степень "магистр")

МЕТОДИЧЕСКИЕ УКАЗАНИЯ
по дисциплине
"Архитектура распределенных программных систем"

Разработчик:
Г.И. Радченко, кандидат физ.-мат. наук

Челябинск-2021

Оглавление

1 Задание 1. Клиент-серверный чат на основе сокетов

Цель: на языке высокого уровня (Java, C#, Python и др. – на выбор обучающегося) реализовать сетевое клиент-серверное приложение – чат (в виде консольного либо диалогового приложения) на основе технологии сокетов. Клиентская программа предоставляет пользователю интерфейс ввода имени пользователя и сообщений, отправляет сообщения серверу и получает от сервера сообщения от остальных пользователей. Соответственно, серверная программа принимает от пользователей сообщения и рассылает их по всем участникам чата, с указанием имени пользователя, который отправил сообщение.

1.1 Критерии оценивания

№	Задача	**Баллы**
1.	Реализовать базовое клиент-серверное приложение «Hello World» на основе сокетов.	5
2.	Реализовать клиентское и серверное приложение для чата с самим собой. *Клиент обеспечивает:* 1) Установку имени пользователя; 2) Подключение к серверу по сетевому имени/адресу; 3) Отправку текстовых сообщений от имени пользователя; 4) Получение и отображение сообщений от сервера; 5) Отключение от сервера. *Сервер обеспечивает:* 1) Подключение одного пользователя; 2) Получение сообщений от пользователя и отправку их обратно; 3) Отключение пользователя от сервера.	5
3.	Модифицировать сервер таким образом, чтобы при соединении и разрыве соединения в окно чата всем пользователям отображалось время, логин и сообщение о том что такой-то пользователь «вошел/вышел»	2

4.	Сдача в срок: задачи 1-3 сданы до 01 марта – 2 балла; до 15 марта – 1 балл; после 15 марта – 0 баллов.	2
5.	Модифицировать сервер таким образом, чтобы он обеспечивал одновременной общение 2-х и более пользователей одновременно в одной общей комнате. Клиент и Сервер должны поддерживать вывод списка подключенных пользователей.	3
6.	Модифицировать клиент и сервер таким образом, чтобы они обеспечивали возможность работы с «Комнатами». Пользователи должны иметь возможность создавать именованные комнаты, изолированные от остальных. Сообщения должны направляться в конкретную комнату чата, и пересылаться только пользователям, подключенным к конкретной комнате.	3
7*	**Реализовать возможность обмена файлами в рамках комнаты. Пользователь должен иметь возможность загрузить файл на сервер в комнату. В этом случае, все участники чата, подключенные к данной комнате должны иметь возможность загрузить данный файл. Через определенный промежуток времени файл должен удаляться с сервера.**	**4***
8*	**Реализовать возможность децентрализованного обмена файлами. Файл не загружается на сервер, но формируется специального рода ссылка, обеспечивающая возможность любому участнику чата напрямую загрузить файл с клиента-отправителя, пока клиент подключен к чату и пока на нем доступен данный файл.**	**4***

1.2 Методические указания

Сокет - конечная точка связи двустороннего канала между 2 процессами, выполняющимися либо на одном, либо на разных компьютерах, соединенных сетью. При соединении 2-х сокетов образуется канал, через который можно передавать данные в обе стороны. Одна сторона канала называется **сервером**, другая - **клиентом**. Существует 2 вида сокетов: *потоковые и дейтаграммные*.

Потоковые сокеты работают с установкой соединения, обеспечивая надежную идентификацию обоих сторон, гарантируют целостность и успешность доставки данных. Основываются на протоколе TCP.

Дейтаграмные сокеты работают без установки соединения и не обеспечивают ни идентификации отправителя, ни контроля успешности доставки данных. Ввиду этого они заметно быстрее потоковых. Основываются на протоколе UDP.

Сокеты связываются между собой через порты.

1.3 Примеры реализации

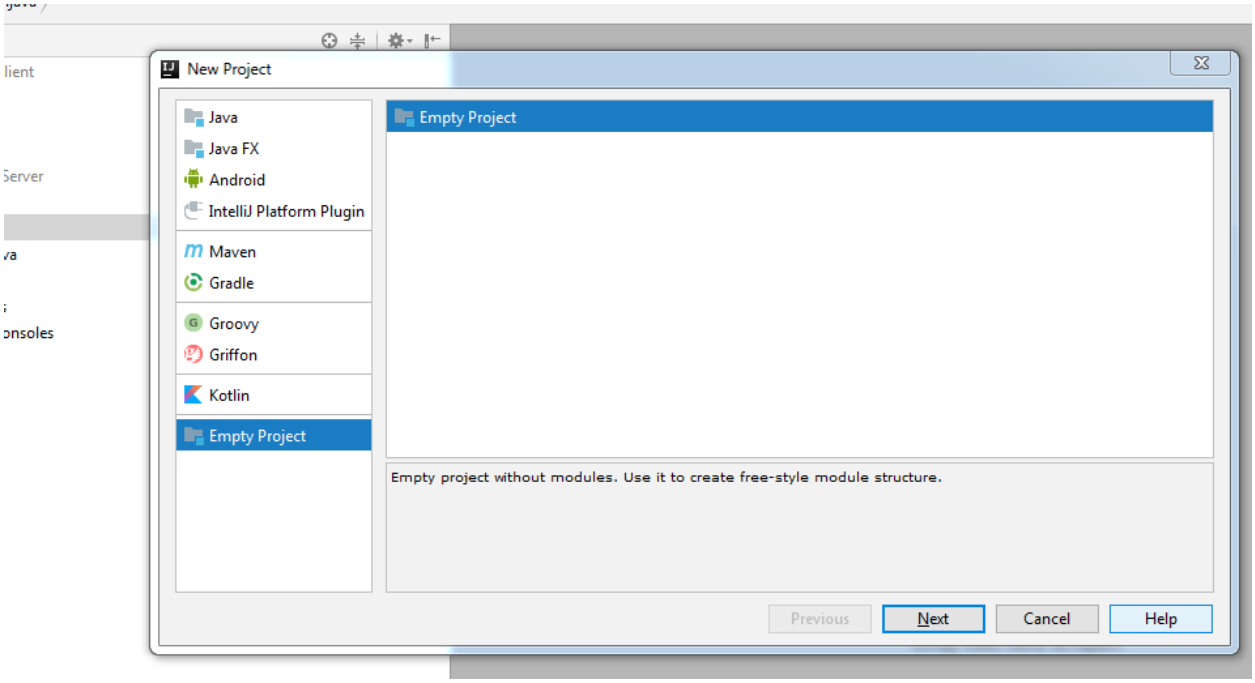
Реализация простого клиент-серверного приложения. Клиент отправляет сообщение серверу и выводит ответ от сервера. Сервер выводит сообщение, полученное от клиента, и отправляет его обратно.

1.3.1 Пример реализации на языке Java

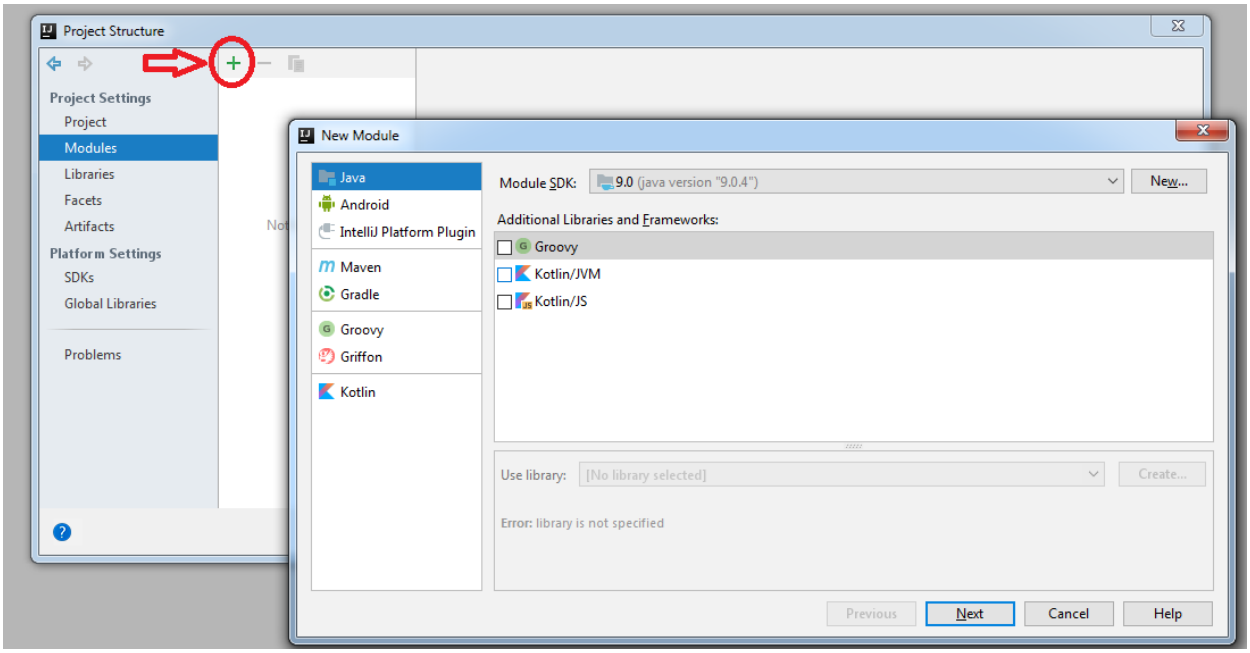
Необходимо загрузить и установить последнюю Java SDK.
IntelliJ IDEA Community Edition вы можете загрузить по ссылке.

Пример реализации клиент-серверного приложения с использованием технологии сокетов на языке Java:

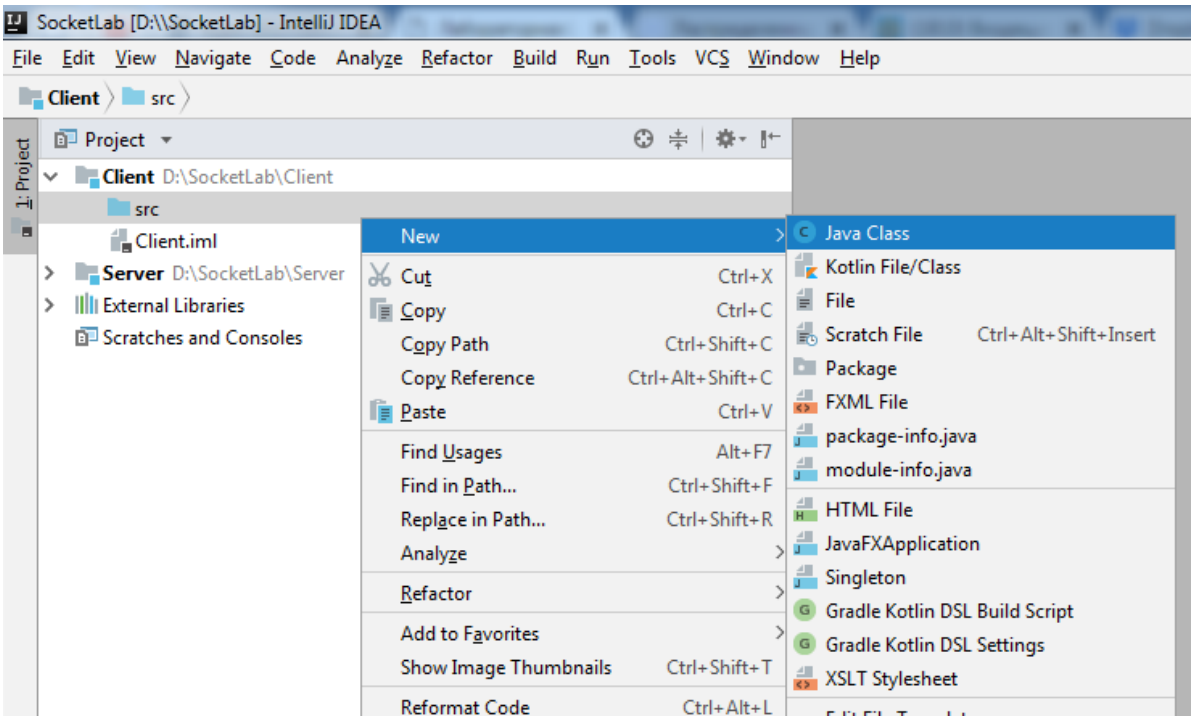
1) В среде IntelliJ IDEA создайте новый проект типа «Empty Project» с названием “SocketLab”.



2) После создания проекта откроется окно «Структура проекта». Добавьте два новых модуля с названиями “Server” и “Client”.



3) В директории «src» модуля “Server” создайте .java файл с именем “Server”. Аналогично для модуля “Client” с именем “Client”.



4) Реализуем исходный код сервера:

a. Подключим библиотеки для работы с сокетами:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
```

```
import java.net.ServerSocket;
import java.net.Socket;
```

b. Укажем порт, подключение на который будет ожидать сервер. Для сервера хост указывать не нужно, он по умолчанию поднимается на “localhost”:

```
public static final int PORT = 19000;
```

c. В функции main запустим сервер и будем ожидать подключение клиента:

```
public static void main(String[] args)  {
    ServerSocket serverSocket = null;

    try {
        serverSocket = new ServerSocket(PORT);

        Socket socket = serverSocket.accept();

    } catch (IOException e){
    e.printStackTrace();
    }
}
```

d. После подключения клиента читаем отправленные им данные и выводим их на экран:

```
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();

byte[] buf = new byte[32*1024];
int readBytes = in.read(buf);
String line = new String(buf, 0, readBytes);
System.out.printf("Client> %s", line);
```

e. После чего отправляем сообщение обратно клиенту:

```
out.write(line.getBytes());
out.flush();
```

В итоге простейший сокет-сервер будет выглядеть следующим образом:

```
public class Server {

    public static final int PORT = 19000;

    public static void main(String[] args)  {
        ServerSocket serverSocket = null;
        try {
            serverSocket = new ServerSocket(PORT);

            System.out.println("Started, waiting for connection");

            Socket socket = serverSocket.accept();

            System.out.println("Accepted. " + socket.getInetAddress());

            InputStream in = socket.getInputStream();
```

```
            OutputStream out = socket.getOutputStream();

            byte[] buf = new byte[32*1024];
            int readBytes = in.read(buf);
            String line = new String(buf, 0, readBytes);
            System.out.printf("Client> %s", line);

            out.write(line.getBytes());
            out.flush();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

5) Реализуем исходный код клиента:

   a. Исходный код клиента будет похож на исходный код сервера, за исключением явного объявления хоста для сокета:

```
public static final int PORT = 19000;
public static final String HOST = "localhost";
```

и блока отправки сообщения:

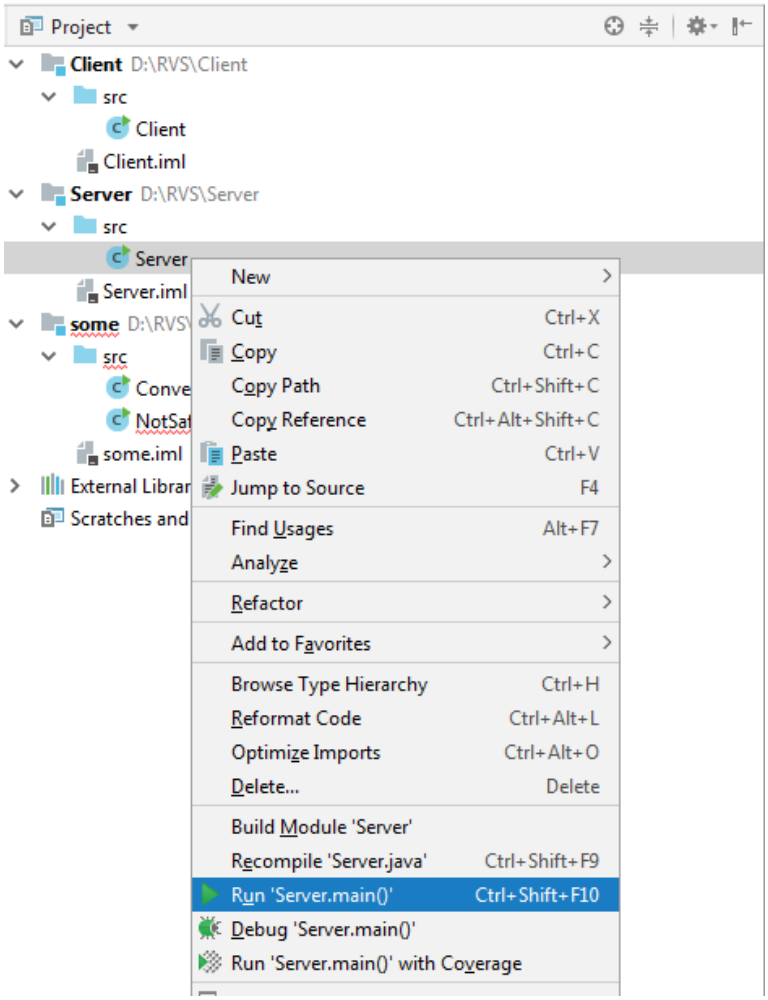
```
try (InputStream in = socket.getInputStream();
     OutputStream out = socket.getOutputStream()) {

    String line = "Hello!";
    out.write(line.getBytes());
    out.flush();
}
```

В итоге простейший сокет-клиент будет выглядеть следующим образом:

```
public class Client {

    public static final int PORT = 19000;
    public static final String HOST = "localhost";

    public static void main(String[] args) {
        Socket socket = null;
        try {
            socket = new Socket(HOST, PORT);

            try (InputStream in = socket.getInputStream();
                 OutputStream out = socket.getOutputStream()) {

                String line = "Hello!";
                out.write(line.getBytes());
                out.flush();

                byte[] buf = new byte[32*1024];
                int readBytes = in.read(buf);

                System.out.printf("Server> %s", new String(buf, 0,
readBytes));
```

```
            }
        } catch (IOException e){
            e.printStackTrace();}
    }
}
```

6) После того, как исходный код клиента и сервера подготовлен, можно запустить приложение. Для этого можно воспользоваться интерфейсом среды IntelliJ IDEA. Нам необходимо запустить в начале сервер, после чего запустить приложение-клиент. Это можно сделать, нажав правой кнопкой на класс «Server» и выбрав: «Run ‘Server.main()’» или сочетанием клавиш Ctrl + Shift + F10



После нажатия, запустится сервер и перейдет в режим ожидания соединения.
После этого, аналогичным образом, необходимо запустить клиентскую часть приложения.

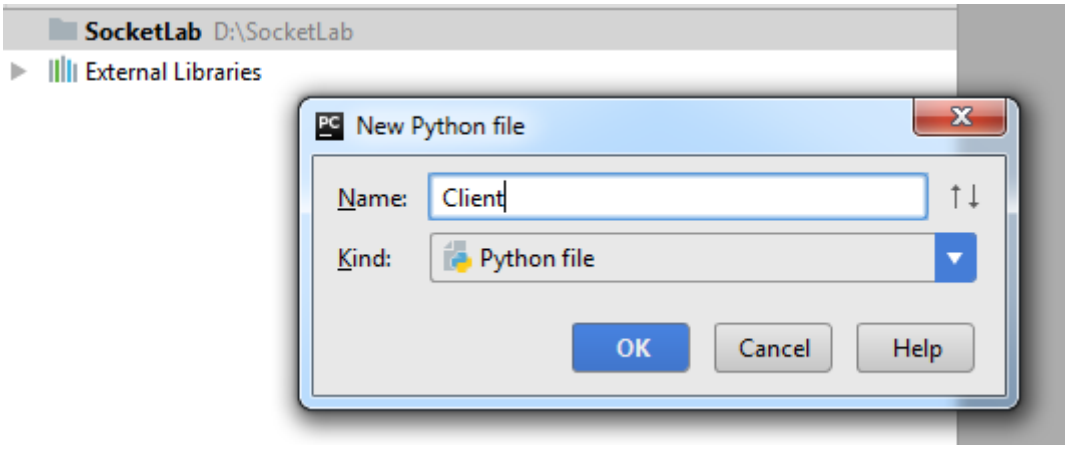
1.3.2 Пример реализации на языке Python

PyCharm Community Edition вы можете загрузить по ссылке.

Пример реализации клиент-серверного приложения с использованием технологии сокетов на языке Python:

1) В среде PyCharm создайте новый проект с названием “SocketLab”.

2) После создания проекта добавьте два Python файла с названиями “Server” и “Client”.



3) Реализуем исходный код сервера:

a. Подключим библиотеку для работы с сокетами:

```
import socket
```

b. Создадим сокет:

```
sock = socket.socket()
```

c. Теперь свяжем наш сокет с хостом и портом с помощью метода bind, которому передается кортеж, первый элемент которого — хост, а второй — порт:

```
sock.bind( ('', 9090) )
```

Насчет хоста — мы оставим строку пустой, чтобы наш сервер был доступен для всех интерфейсов.

d. С помощью метода listen мы запустим для данного сокета режим прослушивания. Метод принимает один аргумент — максимальное количество подключений в очереди:

```
sock.listen(1)
```

e. Теперь мы можем принять подключение с помощью метода accept, который возвращает кортеж с двумя элементами: новый сокет и адрес клиента:

```
conn, addr = sock.accept()
```

f. Для чтения данных используется функция recv, которой первым параметром нужно передать количество получаемых байт данных:

```
data = conn.recv(1024)
```

Тип возвращаемых данных — bytes. У этого типа есть почти все методы, что и у строк, но для того, чтобы использовать из него текстовые данные с другими строками, придётся декодировать данные и использовать уже полученную строку

```
uData = data.decode("utf-8")
print("Client > " + uData)
```

g. Для отправки данных в сокет используется функция send. Принимает она тоже bytes, поэтому для отправки строки вам придётся её закодировать:

```
conn.send(uData.encode("utf-8"))
```

h. После всего и клиенту, и серверу необходимо закрыть сокет с помощью функции close:

```
conn.close()
```

В итоге простейший сокет-сервер будет выглядеть следующим образом:

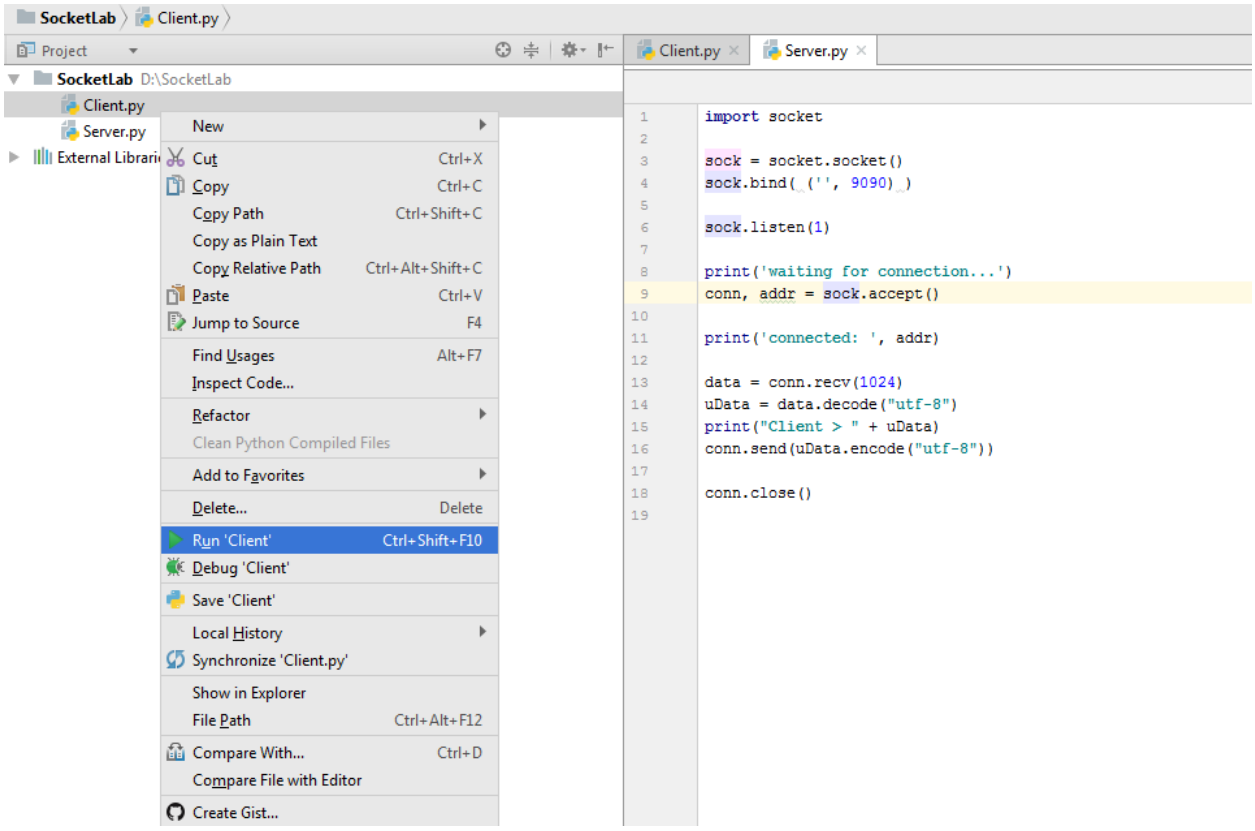
```
import socket

sock = socket.socket()
sock.bind( ('', 9090) )

sock.listen(1)

print('waiting for connection...')
conn, addr = sock.accept()

print('connected: ', addr)

data = conn.recv(1024)
uData = data.decode("utf-8")
print("Client > " + uData)
conn.send(uData.encode("utf-8"))

conn.close()
```

4) Реализуем исходный код клиента:

a. Исходный код клиента будет похож на исходный код сервера:

```
import socket

sock = socket.socket()
sock.connect(('localhost', 9090))

sock.send(b'Hello!\n')

data = sock.recv(1024)
udata = data.decode("utf-8")
print("Server > " + udata)

sock.close()
```

5) После того, как исходный код клиента и сервера подготовлен, можно запустить приложение. Для этого необходимо запустить в начале сервер, после чего запустить клиент. Это можно сделать, нажав правой кнопкой на файл «Server.py» и выбрав: «Run ‘Server’» или сочетанием клавиш Ctrl + Shift + F10

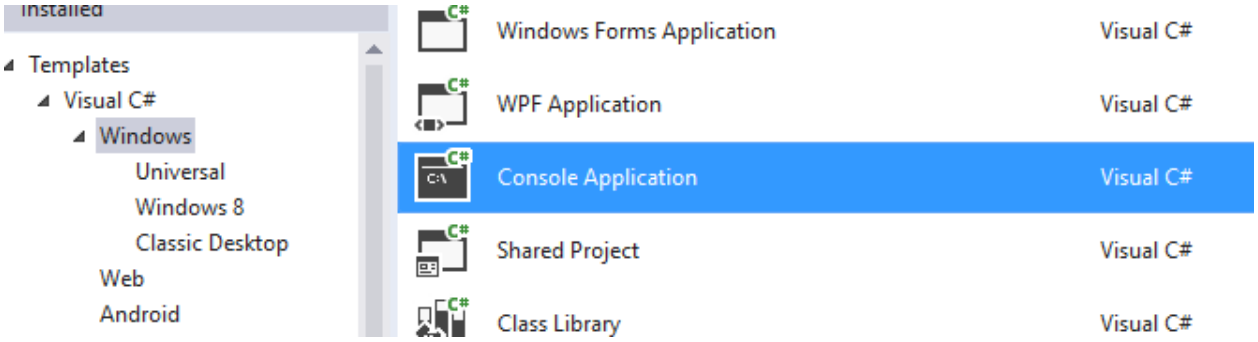


После нажатия, запустится сервер и перейдет в режим ожидания соединения.
После этого, аналогичным образом, необходимо запустить клиентскую часть приложения.

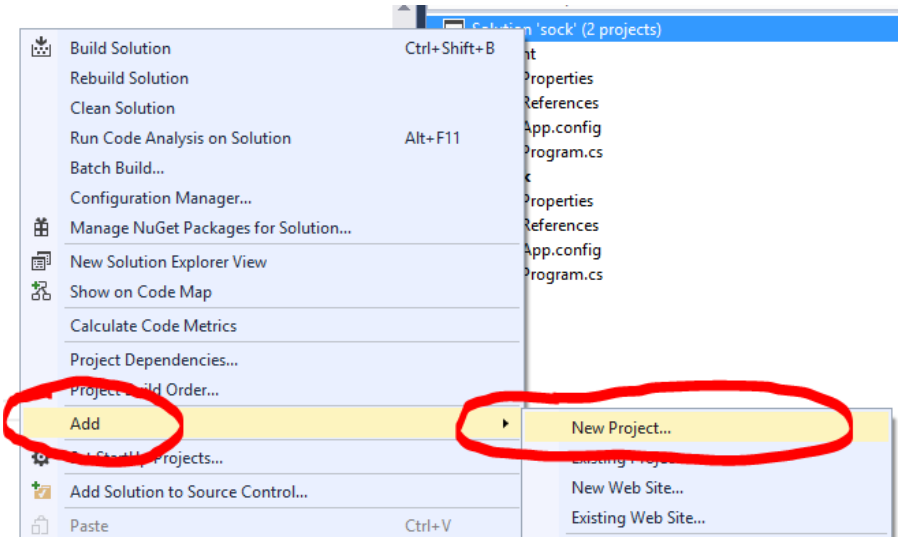
1.3.3 Пример реализации на языке C#

Пример реализации клиент-серверного приложения с использованием технологии сокетов на языке C#:

1) В среде Visual Studio создайте новый проект типа «Консольное приложение» под названием “SocketLab”.



2) В созданном решении будет создан один проект по умолчанию. Для клиент-серверного приложения нам необходимо создать 2 независимых проекта (один – для клиента, другой – для сервера). Таким образом, добавим еще один проект в наше решение, нажав правой кнопкой по решению, и выбрав «Добавить…-> Новый проект»



В открывшемся окне, создадим еще одно консольное приложение, назвав его “SocketClient”.

3) Реализуем исходный код сервера:

   a. В блок using необходимо добавить библиотеки для работы с сокетами:

```
using System.Net;
using System.Net.Sockets;
```

b. В функцию main добавим информацию о конечной точке нашего сервера:

```
Int32 serverPort = 13000;
IPAddress localAddr = IPAddress.Parse("127.0.0.1");
TcpListener calcServer = new TcpListener(localAddr, serverPort);
```

c. Запустим сервер и будем ожидать подключения клиента:

```
calcServer.Start();
TcpClient client = calcServer.AcceptTcpClient();
NetworkStream stream = client.GetStream();
```

d. После подключения клиента, считаем отправленные им данные и выведем их на экран:

```
Byte[] bytes = new Byte[256];
string data = null;
int i = stream.Read(bytes, 0, bytes.Length);
data = System.Text.Encoding.UTF8.GetString(bytes, 0, i);

Console.Write(data);
```

В итоге, основной метод простейшего сокет-сервера может выглядеть следующим образом:

```
            Int32 serverPort = 13000;
            IPAddress localAddr = IPAddress.Parse("127.0.0.1");
            TcpListener calcServer = new TcpListener(localAddr, serverPort);

            calcServer.Start();
            TcpClient client = calcServer.AcceptTcpClient();
            NetworkStream stream = client.GetStream();

            Byte[] bytes = new Byte[256];
            string data = null;
            int i = stream.Read(bytes, 0, bytes.Length);
            data = System.Text.Encoding.UTF8.GetString(bytes, 0, i);

            Console.Write(data);
            Console.ReadKey();
```

4) Реализуем исходный код клиента:

a. Исходный код клиента будет напоминать исходный код сервера, за исключением блока подключения к серверу:

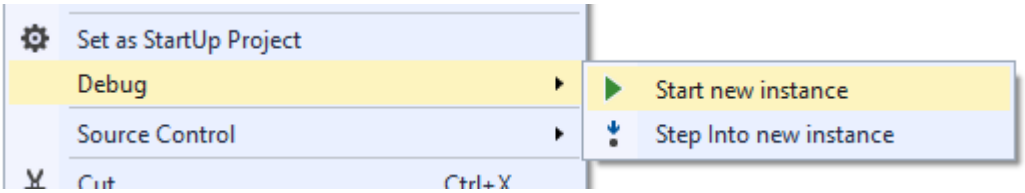
```
…
TcpClient client = new TcpClient();
client.Connect(serverAddr, serverPort);
NetworkStream stream = client.GetStream();
…
```

и блока отправки сообщения:

```
…
String data = "Socket Hello World";
bytes = System.Text.Encoding.UTF8.GetBytes(data);
stream.Write(bytes, 0, bytes.Length);

client.Close();
…
```

По данным указаниям, вам предлагается самостоятельно реализовать простейший сокет-клиент, отправляющий одиночное сообщение на сервер.

5) После того, как исходный код клиента и сервера подготовлен и собран, можно запустить приложение. Для этого можно воспользоваться интерфейсом среды Visual Studio. Нам необходимо запустить в начале сервер, после чего запустить приложение-клиент. Это можно сделать, нажав правой кнопкой на проект «SocketLab» и выбрав: «Отладка->Запуск нового экземпляра»



После нажатия, запустится сервер и перейдет в режим ожидания соединения.

После этого, аналогичным образом, необходимо запустить клиентскую часть приложения, нажав правой кнопкой по проекту «SocketClient» и выбрав аналогичный пункт меню. Альтернативно, клиент и сервер можно запустить непосредственно из папки проекта, запустив исполняемые файлы SocketLab.exe и SocketClient.exe из директорий bin\Debug соответствующих проектов.

1.4 Задание на самостоятельную работу

На основе разработанного простейшего приложения с использованием технологии сокетов, вам необходимо реализовать сетевое клиент-серверное приложение – чат (в виде консольного либо диалогового приложения) на основе технологии сокетов.

1.5 Ссылки

Пространство имен System.Net.Sockets (C#)
Сокеты в Python

Задание 2. Исследование методов сериализации данных

Цель: на языке высокого уровня (Java, C#, Python и др. – на выбор обучающегося) реализовать приложение для тестирования эффективности работы с различными форматами сериализации данных (http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats). В процессе тестирования форматов сериализации необходимо учитывать следующие характеристики:

1. Размер сериализованной структуры данных;
2. Время сериализации/десериализации.

В сериализуемой структуре желательно представить несколько различных видов данных, включая:

- строковые данные,
- массивы данных,
- словари,
- целочисленные данные,
- данные с плавающей запятой.

Для отчета необходимо представить:

1) Отдельное приложение для тестирования форматов сериализации;
2) Отчет в формате таблицы Excel о форматах сериализации данных.

1.6 Критерии оценивания

№	Задача	**Баллы**
1.	Протестировать нативный вариант сериализации (в зависимости от языка)	4
2.	Протестировать сериализацию в XML	4
3.	Протестировать сериализацию в JSON	4
4.	Сдача в срок: задачи 1-3 сданы до 29 марта – 2 балла; до 12 апреля – 1 балл; после 12 апреля – 0 баллов.	2
5.	Протестировать сериализацию в Google Protocol Buffers	3
6.	Протестировать сериализацию в Apache Avro	3
7*	**Протестировать сериализацию в YAML**	**2***

8*	**Протестировать сериализацию в MessagePack**	**2***

1.7 Методические указания

Тестирование проводится на основе структуры данных, хранящейся в оперативной памяти в виде объектов или структур. Приложение должно выводить на консоль информацию о следующих характеристиках данных:

1. Процесс сохранения данных из структуры в сериализованную структуру в оперативной памяти:
   1.1. Полученный объем
   1.2. Время сериализации
2. Процесс загрузки данных из файла на жестком диске в оперативную память:
   2.1. Время десериализации

Обычно, тестирование времени сериализации и десериализации замеряется как среднее время при выполнении большого числа операций сериализации/десериализации (порядка 1000 циклов). Для учета времени выполнения процедуры используются различные механизмы профилирования времени выполнения кода, в зависимости от используемого языка:
1) C++: http://esate.ru/blog/cpp/364.html
2) C#: http://msdn.microsoft.com/ru-ru/library/system.diagnostics.stopwatch.aspx
3) Python: http://john16blog.blogspot.ru/2012/02/python-timeit.html
4) Java: http://www.javaspecialist.ru/2012/04/system.html

В сериализуемой структуре данных желательно представить несколько различных видов данных, включая: строковые данные, массивы данных, словари, целочисленные данные, данные с плавающей запятой.

1.8 Примеры реализации

1.8.1 Пример реализации на языке Python

Пример генерации структуры для сериализации (Python):

```
d = {
   'words': """
        Lorem ipsum dolor sit amet, consectetur adipiscing
        elit. Mauris adipiscing adipiscing placerat.
        Vestibulum augue augue,
        pellentesque quis sollicitudin id, adipiscing.
```

```
        """,
    'list': range(100),
    'dict': dict((str(i),'a') for i in xrange(100)),
    'int': 100,
    'float': 100.123456
}
```

Можно использовать данный вариант тестирования Native Serialization и JSON на Питоне как пример реализации таких тестов.

```
"""
Dependencies:

    pip install tabulate simplejson

"""

from timeit import timeit
from tabulate import tabulate
import sys

message = '''d = {
    'PackageID' : 1539,
    'PersonID' : 33,
    'Name' : """MEGA_GAMER_2222""",
    'Inventory': dict((str(i),i) for i in xrange(100)),
    'CurrentLocation': """
            Pentos is a large port city, more populous than Astapor on Slaver Bay,
            and may be one of the most populous of the Free Cities.
            It lies on the bay of Pentos off the narrow sea, with the Flatlands
            plains and Velvet Hills to the east.
            The city has many square brick towers, controlled by the spice
traders.
            Most of the roofing is done in tiles. There is a large red temple in
            Pentos, along with the manse of Illyrio Mopatis and the Sunrise Gate
            allows the traveler to exit the city to the east,
            in the direction of the Rhoyne.
            """
}'''

setup_pickle    = '%s ; import pickle ; src = pickle.dumps(d, 2)' % message
setup_json      = '%s ; import json; src = json.dumps(d)' % message

tests = [
    # (title, setup, enc_test, dec_test)
    ('pickle (native serialization)', 'import pickle; %s' % setup_pickle,
'pickle.dumps(d, 2)', 'pickle.loads(src)'),
    ('json', 'import json; %s' % setup_json, 'json.dumps(d)', 'json.loads(src)'),
]

loops = 5000
enc_table = []
dec_table = []
```

```
print "Running tests (%d loops each)" % loops

for title, mod, enc, dec in tests:
    print title

    print "  [Encode]", enc
    result = timeit(enc, mod, number=loops)
    exec mod
    enc_table.append([title, result, sys.getsizeof(src)])

    print "  [Decode]", dec
    result = timeit(dec, mod, number=loops)
    dec_table.append([title, result])

enc_table.sort(key=lambda x: x[1])
enc_table.insert(0, ['Package', 'Seconds', 'Size'])

dec_table.sort(key=lambda x: x[1])
dec_table.insert(0, ['Package', 'Seconds'])

print "\nEncoding Test (%d loops)" % loops
print tabulate(enc_table, headers="firstrow")

print "\nDecoding Test (%d loops)" % loops
print tabulate(dec_table, headers="firstrow")
```

1.9 Задание на самостоятельную работу

Далее, в зависимости от выбранной вами технологии и языка программирования, вам предлагается самостоятельно провести исследование методов сериализации данных и подготовить краткий отчет.

1.10 Ссылки

Для дальнейшего самостоятельного изучения данной темы можно воспользоваться следующими ресурсами:

1. Comparison of data serialization formats. http://en.wikipedia.org/wiki/Comparison_of_data_serialization_formats
2. Serialization. http://en.wikipedia.org/wiki/Data_serialization (в статье приводится информация об основных методах реализации сериализации в различных средах программирования).

Задание 3. Разработка веб-сервиса на основе gRPC

Цель: на языке высокого уровня (Java, C#, Python, Go и др. – на выбор обучающегося) реализовать gRPC веб-сервис и клиента для него, которые бы обеспечивали функционирование социальной сети. Сервис должен предоставлять API для просмотра ленты сообщений, в которую пользователи могут загружать сообщения из своих клиентов. Другие клиенты могут «лайкать» чужие посты и оставлять к ним комментарии.

1.11 Критерии оценивания

№	Задача	**Баллы**
1.	Реализовать и разместить локально простейший gRPC-сервис «Reverse» который принимает от клиента строку и возвращает ее в обратном порядке.	5
2.	Реализовать и разместить локально gRPC веб-сервис, и клиента для социальной сети обеспечивающий сбор и отображение ленты сообщений от пользователей.	5
3.	Сдача в срок: задачи 1-2 сданы до 26 апреля – 2 балла; до 9 мая– 1 балл; после 9 мая – 0 баллов.	2
4.	Реализовать и разместить локально gRPC веб-сервис, и клиента для социальной сети обеспечивающий возможность лайкать сообщения и оставлять комментарии к чужим постам.	4
5.	Разместить разработанный gRPC-сервис в облаке и продемонстрировать его работу.	4
6*	**Реализовать функционал прямой отправки текстовых сообщений от одного пользователя вашей социальной сети другому**	**5**

1.12 Методические указания

Рассмотрим пример реализации gRPC веб-сервиса, обеспечивающего «разворот» входящий строки текста на языке Go (на основе статьи https://habr.com/ru/post/461279/)

1.13 Примеры реализации

1.13.1 Разработка gRPC сервиса на основе Go

1.13.2 .proto-файл

.proto-файл описысывает, какие операции наш сервис будет осуществлять и какими данными он при этом будет обмениваться. Создаем в проекте папку ***proto***, а в ней — файл ***reverse.proto***

```
syntax = "proto3";

package reverse;

service Reverse {
    rpc Do(Request) returns (Response) {}
}

message Request {
    string message =1;
}

message Response {
    string message =1;
}
```

Функция, которая вызывается удаленно на сервере и возвращает данные клиенту, помечается как ***rpc***. Структуры данных, служащие для обмена информацией между взаимодействующими узлами, помечаются как ***message***. Каждому полю сообщения необходимо присвоить порядковый номер. В данном случае наша функция принимает от клиента сообщения типа ***Request*** и возвращает сообщения типа ***Response***.
Как только мы создали **.proto**-файл, необходимо получить **.go**-файл нашего сервиса. Для этого нужно выполнить седующую консольную команду в папке ***proto***:

```
$ protoc -I . reverse.proto --go_out=plugins=grpc:.
```

Разумеется, сначала вам нужно выполнить сборку gRPC.
Выполнение вышеприведенной команды создаст новый **.go**-файл, содержащий методы для создания клиента, сервера и сообщений, которыми они обмениваются. Если мы вызовем **godoc**, то увидим следующее:

```
$ godoc .
PACKAGE DOCUMENTATION

package reverse
    import "."

Package reverse is a generated protocol buffer package.

It is generated from these files:

reverse.proto

It has these top-level messages:

Request
 Response
....
```

1.13.3 Клиент

Было бы неплохо, если бы наш клиент работал вот так:

```
reverse "this is a test"
```

```
tset a si siht
```

Вот код, который создает **gRPC**-клиент, используя структуры данных, сгенерированные из **.proto**-файла:

```
package main

import (
    "context"
    "fmt"
    "os"
     pb "github.com/matzhouse/go-grpc/proto"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)
```

```
func main() {
    opts := []grpc.DialOption{
        grpc.WithInsecure(),
    }
    args := os.Args
    conn, err := grpc.Dial("127.0.0.1:5300", opts...)

    if err != nil {
        grpclog.Fatalf("fail to dial: %v", err)
    }

    defer conn.Close()

    client := pb.NewReverseClient(conn)
    request := &pb.Request{
        Message: args[1],
    }
    response, err := client.Do(context.Background(), request)

    if err != nil {
        grpclog.Fatalf("fail to dial: %v", err)
    }

   fmt.Println(response.Message)
}
```

1.13.4 Сервер

Сервер использует тот же самый сгенерированный **.go**-файл. Однако он определяет только интерфейс сервера, логику же нам придется реализовать самостоятельно:

```
package main

import (
    "net"
    pb "github.com/matzhouse/go-grpc/proto"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

func main() {
    listener, err := net.Listen("tcp", ":5300")

    if err != nil {
        grpclog.Fatalf("failed to listen: %v", err)
    }

    opts := []grpc.ServerOption{}
    grpcServer := grpc.NewServer(opts...)
```

```
    pb.RegisterReverseServer(grpcServer, &server{})
    grpcServer.Serve(listener)
}

type server struct{}

func (s *server) Do(c context.Context, request *pb.Request)
(response *pb.Response, err error) {
    n := 0
    // Create an array of runes to safely reverse a string.
    rune := make([]rune, len(request.Message))

    for _, r := range request.Message {
        rune[n] = r
        n++
    }

    // Reverse using runes.
    rune = rune[0:n]

    for i := 0; i < n/2; i++ {
        rune[i], rune[n-1-i] = rune[n-1-i], rune[i]
    }

    output := string(rune)
    response = &pb.Response{
        Message: output,
    }

    return response, nil
}
```

После того, как мы подготовили исходный код сервера, мы можем запустить его следующей командой

```
$ go build -o reverse
$ ./reverse "this is a test"
tset a si siht
```

1.14 Задание на самостоятельную работу

На основе представленных примеров вам предлагается самостоятельно реализовать gRPC веб-сервис, обеспечивающий решение задач 1-5.

1.15 Ссылки

Для дальнейшего самостоятельного изучения данной темы можно воспользоваться следующими ресурсами:

3. Руководство для начинающих в платформе Amazon Web Services доступны по ссылке: https://aws.amazon.com/ru/getting-started/?nc1=h_ls.
4. Как создать простой микросервис на Golang и gRPC и выполнить его контейнеризацию с помощью Docker https://habr.com/ru/post/461279/
5. Создание gRPC сервиса на Node.js https://codelabs.developers.google.com/codelabs/cloud-grpc-ru/index.html?index=..%2F..lang-ru#0

Задание 4. REST-сервис с асинхронной обработкой запросов

Цель: на языке высокого уровня (Java, C#, Python, Go и др. – на выбор обучающегося) реализовать REST веб-сервис, принимающий на обработку CSV-файлы и *реализующий их асинхронную (отложенную) обработку*. Отложенное выполнение задач необходимо для того, чтобы обеспечить быстрый отклик в веб-приложениях и должно применяться во всех потенциально длительных операциях (посылка писем, обработка файлов и т.д.)

Веб-сервис должен предоставлять веб-API, обеспечивающий загрузку в сервис таблицы в виде CSV-файла для ее обработки и получения результата в виде N элементов из данного файла с максимальным значением значения данных по определенному полю данного CSV-файла (TOP-N). Первая строка CSV-файла содержит заголовки соответствующих столбцов таблицы. В качестве примера можно использовать данные со страницы https://support.spatialkey.com/spatialkey-sample-csv-data/ (например, http://samplecsvs.s3.amazonaws.com/Sacramentorealestatetransactions.csv).

1.1 Критерии оценивания

№	Задача	**Баллы**
1.	Реализовать и разместить локально простейший REST-сервис «Эхо» по адресу `example.edu/echo` принимающий строку текста по PUT-запросу и возвращающий эту же строку текста по GET-запросу	5
2.	Реализовать и разместить локально веб-сервис, предоставляющий по адресу `example.edu/top` веб-API для загрузки файлов с параметрами: – `field` - поле, по которому нужно сортировать записи – `count` - количество верхних записей соответственно. В ответ на запрос должна возвращаться ссылка на результирующий файл, в который будут добавлены результаты обработки.	5

3.	Реализовать метод `example.edu/upload`, обеспечивающий загрузку, асинхронную обработку записей из CSV-файла и добавление их в БД. По запросу `GET example.edu /top?field=FIELD&count=N` должен возвращаться JSON-ответ, содержащий верхние N записей по полю FIELD.	5
4.	Разместить разработанный REST-сервис в облаке и продемонстрировать его работу.	5
5*	**Обеспечить распределенную обработку CSV-файла из задания 2. Файл должен разбиваться на части, и отдаваться на независимую обработку набору исполнителей. Каждый из исполнителей осуществляет независимый разбор файла и находит N верхних записей для своего сегмента, затем результаты обработки объединяются и берутся верхние N записей от результата (реализовать концепцию "MAP-REDUCE" в ручном режиме).**	**8**

1.2 Методические указания

Для реализации данного задания необходимо воспользоваться очередью, обеспечивающей асинхронный обмен сообщениями между компонентами системы. Рассмотрим ключевые подходы к реализации очередей сообщений на платформах Redis, RabbitMQ и Apache Kafka.

1.3 Примеры реализации

1.3.1 Использование системы Redis на базе Java

Как одно из возможных решений, можно воспользоваться системой Redis: http://redis.io. Для выполнения лабораторной работы можно как самостоятельно развернуть данную систему, так и воспользоваться готовым облачным решением, например http://redistogo.com/.

Для реализации веб-сервиса, легче всего воспользоваться одним из готовых фреймворков для вашего языка программирования. Для языков Python, Ruby и Go можно рекомендовать такие платформы как Flask, Sinatra и Martini соответственно (https://realpython.com/blog/python/python-ruby-and-golang-a-web-Service-application-comparison/).

Рассмотрим пример реализации простейшего веб-сервиса, реализующего асинхронный процесс обработки загруженных файлов, на базе фреймворка Spark для языка Java.

1. Для реализации асинхронного обмена и отложенного задания необходимо завести аккаунт на http://redistogo.com/ (внизу есть маленькая кнопочка free plan), после чего вам выдадут connection string вида redis://redistogo:34534987987@angelfish.redistogo.com:10649/
2. Создадим новый Java-проект, который будет состоять из двух независимых компонентов: веб-сервиса и обработчика.
3. Для работы с проектом, нам потребуется внедрить в проект системы для работы с платформой Redis (jedis) и веб-сервис Spark. Это можно реализовать посредством платформы Maven, указав соответствующие зависимости в конфигурационном файле pom.xml. Рассмотрим блок dependencies которые необходимо добавить в pom.xml (полный код pom.xml доступен вот тут: https://gist.github.com/skayred/0c0a9dc57ad8f9fa9744eea49f1fb50a):

```
<dependencies>
    <dependency>
      <groupId>redis.clients</groupId>
      <artifactId>jedis</artifactId>
<version>2.8.1</version>
    </dependency>
```

```
    <dependency>
      <groupId>com.sparkjava</groupId>
      <artifactId>spark-core</artifactId>
      <version>2.3</version>
    </dependency>
</dependencies>
```

Добавление данных строк в pom.xml и пересборка Maven-проекта загрузит необходимые для реализации проекта зависимости.

4. Сервер обеспечивает обработку POST запросов по адресу “\upload”, принимающих один параметр “file”. Полученный в запросе файл сохраняется во временном хранилище, после чего его имя передается в очередь для дальнейшей обработки.

Рассмотрим исходный код веб-сервиса:

```
package ru.susu;
```

```
import redis.clients.jedis.Jedis;
import static spark.Spark.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.nio.*;
import java.nio.file.*;
import java.util.*;


public class AppServer {
    public static void main( String[] args ) {
        // Соединяемся с Редисом
        Jedis jedis = new Jedis("АДРЕС_РЕДИСА");


        // Тут мы используем фремворк Spark, следующий вызов вешает
слушателя на метод POST
        post("/upload", "multipart/form-data", (request, response) ->
{
                // Получаем файл из ПОСТа и сохраняем на диск
                String location = "/oldhome/sk_/projects/tmp/";
// the directory location where files will be stored
                long maxFileSize = 100000000;       // the maximum
size allowed for uploaded files
                long maxRequestSize = 100000000;    // the maximum
size allowed for multipart/form-data requests
                int fileSizeThreshold = 1024;       // the size
threshold after which files will be written to disk


                MultipartConfigElement multipartConfigElement = new
MultipartConfigElement(

location, maxFileSize, maxRequestSize, fileSizeThreshold);

request.raw().setAttribute("org.eclipse.jetty.multipartConfig",
                                            multipartConfigElement);


                String fName =
request.raw().getPart("file").getSubmittedFileName();


                Part uploadedFile = request.raw().getPart("file");
                Path out = Paths.get(location + fName);
                try (final InputStream in =
uploadedFile.getInputStream()) {
                    Files.copy(in, out);
                    uploadedFile.delete();
                }
                // cleanup
                multipartConfigElement = null;
                uploadedFile = null;
```

```
                // После сохранения файла отправляем путь к нему
обработчику
                jedis.rpush("queue", out.toString());


                return "OK";
            });
    }
}
```

5. Реализуем исполнителя

Исполнитель получает адрес файла, который необходимо обработать, после чего выводит содержимое данного файла.

```
package ru.susu;

    import
redis.clients.jedis.Jedis;  import
java.util.*;  import java.io.*;
 import java.nio.file.*;


    public class App
{
     public static void main( String[] args ) {
         // Соединяемся с Редисом
         Jedis jedis = new Jedis("АДРЕС_РЕДИСА");
         List<String> messages = null;
// Ждем сообщения из очереди
         while(true){
           System.out.println("Waiting for a message in the queue");
           messages = jedis.blpop(0,"queue");
           String payload = messages.get(1);
           // Выполняем задачу
processFile(payload);
         }
     }

     // Выводим содержимое файла в консоль
     private static void processFile(String filename) {
try {
             System.out.println(readFile(filename));
         } catch (IOException e) {
         }
     }
        static String readFile(String path) throws IOException {
byte[] encoded = Files.readAllBytes(Paths.get(path));
return new String(encoded);
     }
}
```

1.3.2 Использование системы RabbitMQ на базе Go

RabbitMQ – это брокер сообщений. Его основная цель – принимать и отдавать сообщения. RabbitMQ позволяет взаимодействовать различным программам при помощи протокола AMQP. RabbitMQ является отличным решением для построения SOA (сервис-ориентированной архитектуры) и распределением отложенных ресурсоемких задач. RabbitMQ написан на языке Erlang и базируеться на базе СУБД Mnesia которая также написана на Erlang. Mnesia – это распределённая СУБД реального времени, по своей сути используется для встраиваемых решений и этим похожа на Berkeley DB

Для использования RabbitMQ в программах на языке Go необходимо подключить библиотеку (пакет), использующую протокол AMQP. AMQP — это протокол обмена сообщениями на уровне соединения, который можно использовать для создания кроссплатформенных приложений для обмена сообщениями. Цель этого протокола проста — определить механизм безопасной, надежной и эффективной передачи сообщений между двумя сторонами. Для написания программы в ходе работы будет использован пакет amqp для языка Go.

Рассмотрим реализацию простейшего приложения, использующего очередь RabbitMQ для обмена данными.

1. Для разработки и компиляции программы на языке Go необходимо установить стандартный компилятор языка – gc, а также редактор исходного кода, например Visual Studio Code.
2. Чтобы загрузить и установить Go необходимо скачать компилятор с официального сайта http://golang.org/doc/install.html
3. Для установки Visual Studio Code необходимо скачать установщик с официального сайта https://www.visualstudio.com/ru-ru/products/code-vs.aspx Затем нужно запустить исполняемый файл VSCodeSetup-stable.exe. После этого в меню View-Command Palette в VS Code необходимо найти и скачать расширение для языка Go. Для этого во всплывающей строке нужно ввести ext inst и выбрать пункт Extensions: Install Extension. Из полученного списка нужно выбрать расширение для Go. Далее надо установить пакеты и отладчик. Для установки пакетов нужно открыть любой go-файл и нажать на сообщение "Analysis Tools Missing" в правом нижнем углу. Пакеты установятся автоматически. Для установки отладчика необходимо прописать в командную строку Windows команду

```
go get -u github.com/derekparker/delve/cmd/dlv
```

После этого отладчик установится автоматически.

4. Далее следует установить сервер RabbitMQ. Для этого нужно:

1. Скачать и установить сервер RabbitMQ с официального сайта https://www.rabbitmq.com/install-windows.html. На данной странице для просмотра также доступна инструкция по установке данного сервера.
2. Скачать и установить все пакеты Erlang с официального сайта http://www.erlang.org/download.html
3. После завершения обеих установок необходимо перезагрузить компьютер.

5. После выполнения перезагрузки компьютера необходимо запустить командную строку RabbitMQ. В командной строке RabbitMQ необходимо ввести команду

```
rabbitmq-server -detached
```

Данная команда осуществит запуск сервера. В том случае, если сервер уже запущен, в командной строке будет отображена соответствующая запись. Далее надо ввести команду

```
rabbitmq-plugins enable rabbitmq_management
```

Данная команда осуществит включение плагинов. Далее необходимо перейти по ссылке: http://localhost:15672/ и авторизоваться на открывшейся странице (login: guest, password: guest).

6. Также потребуется установить пакет amqp для языка Go. Для этого в командной строке Windows нужно ввести команду

```
go get github.com/streadway/amqp
```

После установки необходимого ПО можно приступать к разработке и реализации программы.

7. Для демонстрации принципа работы RabbitMQ потребуется написать 2 программы: одна будет отправлять сообщения на сервер, а вторая – загружать их оттуда и выводить в консоль. Первая программа будет называться send.go. В ней нужно подключить необходимые для написания программы пакеты

```
package main

import (
    "fmt"
    "log"
    "os"

    "github.com/streadway/amqp"
)
```

8. Также необходимо написать вспомогательную функцию для проверки возвращаемого значения

```
func failOnError(err error, msg string) {
    if err != nil {
        log.Fatalf("%s: %s", msg, err)
        panic(fmt.Sprintf("%s: %s", msg, err))
```

```
    }
}
```

9. Затем подключиться к брокеру сообщений, находящемуся на локальном сервере. Для подключения к брокеру, находящемуся на другой машине, необходимо заменить localhost на IP-адрес этой машины.

```
    conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/%2f")
    failOnError(err, "Failed to connect to RabbitMQ")
    defer conn.Close()
```

10. Далее нужно создать канал для передачи сообщений на сервер

```
    ch, err := conn.Channel()
    failOnError(err, "Failed to open a channel")
    defer ch.Close()
```

11. Для отправки сообщения необходимо создать очередь, в которую будет отправляться данное сообщение

```
    q, err := ch.QueueDeclare(
        "hello", // name
        false,   // durable
        false,   // delete when unused
        false,   // exclusive
        false,   // no-wait
        nil,     // arguments
    )
    failOnError(err, "Failed to declare a queue")
```

12. Далее следует код, отвечающий за отправку сообщения

```
    if len(os.Args) > 1 {
        for tmp := 1; tmp < len(os.Args); tmp++ {
            body += os.Args[tmp] + " "
        }
    } else {
        body = "message"
    }

    err = ch.Publish(
        "",     // exchange
        q.Name, // routing key
        false,  // mandatory
        false,  // immediate
        amqp.Publishing{
            ContentType: "text/plain",
```

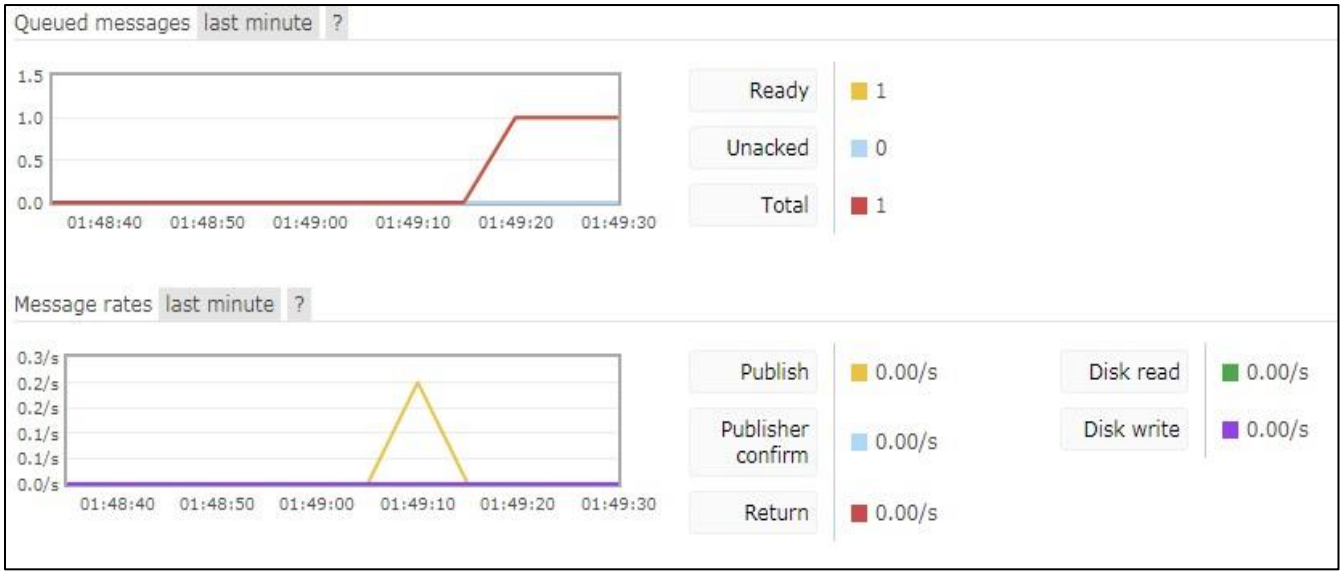
```
            Body:        []byte(body),
        })
    failOnError(err, "Failed to publish a message")
```

13. Теперь необходимо запустить данную программу. Для этого в командной строке Windows нужно перейти в каталог, в котором размещен файл send.go. Затем необходимо написать команду

```
go build
```

которая создаст в данном каталоге исполняемый файл send.exe. Далее в командной строке можно ввести send.exe и сообщение, которое нужно отправить в очередь. Если не указано сообщение, то программа отправит на сервер сообщение "message".

14. После отправки сообщения можно проверить, дошло ли оно, и сколько сообщений находится в очереди в данный момент. Для этого необходимо перейти по ссылке http://localhost:15672/.



15. Далее нужно реализовать программу для получения сообщений с сервера. Исходный файл будет называться receive.go. Нам также потребуется подключить необходимые библиотеки, написать вспомогательную функцию, подключиться к брокеру, создать канал, а также создать очередь (на случай, если получатель будет запущен раньше отправителя). После этого можно выгружать сообщения с сервера из очереди. Так как сервер будет поставлять нам сообщения асинхронно, то мы будем считывать сообщения в горутину.

```
    msgs, err := ch.Consume(
        q.Name, // queue
        "",     // consumer
```

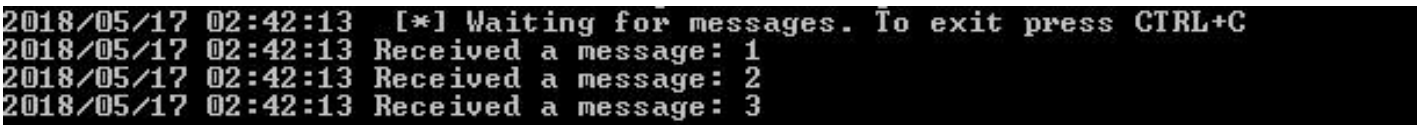
```
        true,   // auto-ack
        false,  // exclusive
        false,  // no-local
        false,  // no-wait
        nil,    // args
    )
    failOnError(err, "Failed to register a consumer")

    forever := make(chan bool)

    go func() {
        for d := range msgs {
            log.Printf("Received a message: %s", d.Body)
        }
    }()

    log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
    <-forever
```

16. Чтобы запустить получателя, нужно так же в командной строке Windows создать и запустить файл receive.exe. После этого в командной строке будут выведены сообщения, которые в данный момент находились в очереди, а получатель продолжит ждать новые сообщения.

17. Для проверки корректности работы попробуем отправить на сервер 3 сообщения: "1", "2", "3".

```
C:\Users\EGOR\Documents\VS Code Projects\Messenger\Send>send.exe 1
C:\Users\EGOR\Documents\VS Code Projects\Messenger\Send>send.exe 2
C:\Users\EGOR\Documents\VS Code Projects\Messenger\Send>send.exe 3
```

18. Далее запускаем приложение-получатель, который должен вывести в консоль соответствующие сообщения.



Получатель вывел сообщения в консоль в нужном порядке, после чего продолжает ждать новые сообщения.

1.4 Задание на самостоятельную работу

На основе представленных примеров вам предлагается самостоятельно реализовать веб-сервис, обеспечивающий решение задач 1-4.

1.5 Ссылки

Для дальнейшего самостоятельного изучения данной темы можно воспользоваться следующими ресурсами:

6. Руководство для начинающих в платформе Amazon Web Services доступны по ссылке: https://aws.amazon.com/ru/getting-started/?nc1=h_ls.
7. Использование асинхронного обмена сообщениями для улучшения доступности https://habr.com/ru/company/piter/blog/458344/
8. Asynchronous Task Execution Using Redis and Spring Boot https://dzone.com/articles/asynchronous-task-executor-using-redis-and-spring
9. Asynchronous Microservices with RabbitMQ and Node.js https://www.manifold.co/blog/asynchronous-microservices-with-rabbitmq-and-node-js