**DOKUMENTACJA**

**PROJEKTU**

**Technologia: REST (Http)**

**Język programowania: Python**

**Skład osobowy grupy:**

* **Kiełbania Piotr**
* **Kowalska Paulina**
* **Kurkowski Wojciech**
* **Machalska Anna**
* **Mazurek Paweł**
* **Pasierbiewicz Karolina**

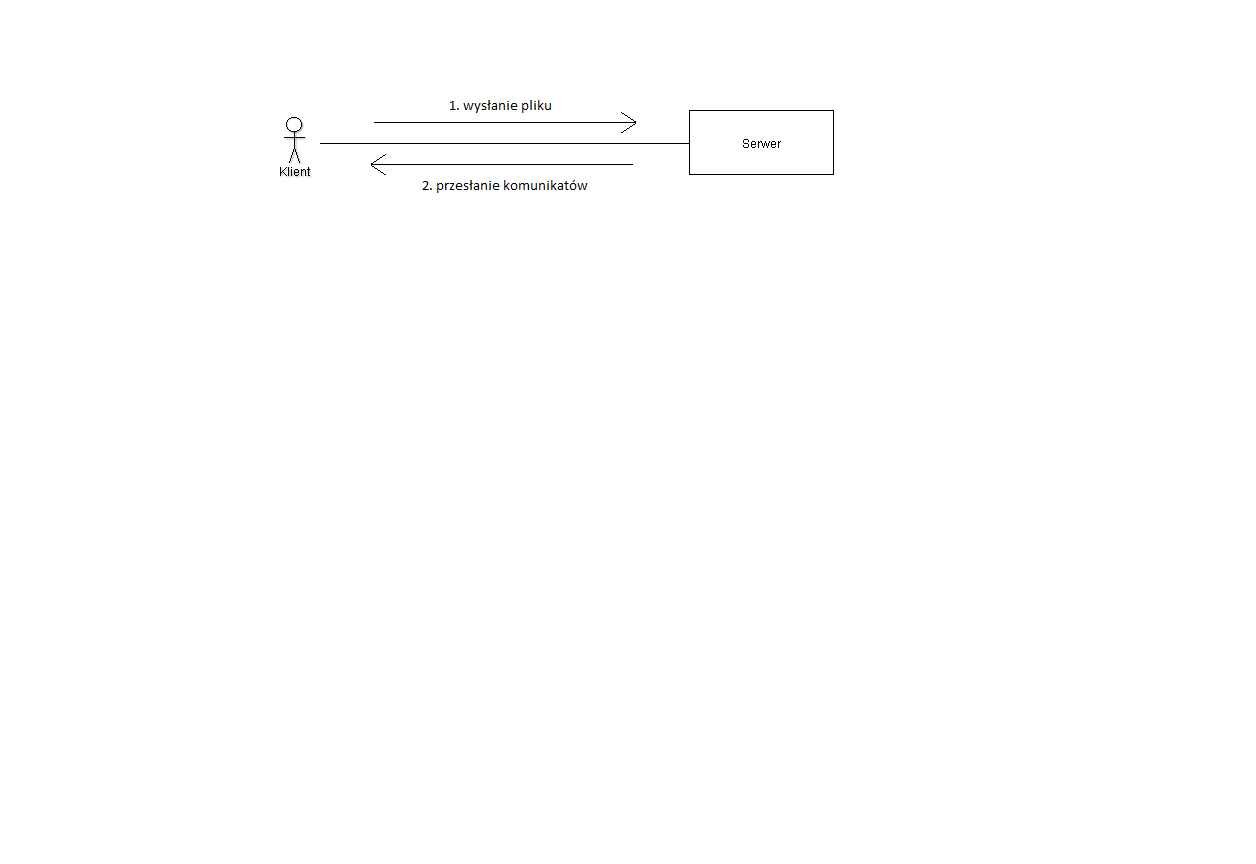
[**https://github.com/paulinakowalska/PythonREST.git**](https://github.com/paulinakowalska/PythonREST.git)

**Analiza dostępności**

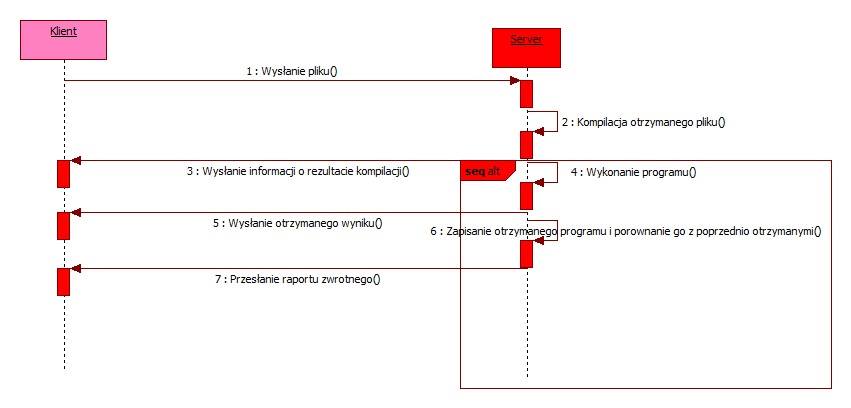
**REST** (ang. **Re**presentationa **S**tate **T**ransfer) jest wzorcem narzucającym dobre praktyki tworzenia architektury aplikacji rozproszonych. Jest jednym z najpopularniejszych technologii wykorzystanych do implementacji wspomnianych aplikacji rozproszonych. Dostępność technologii REST dla wielu języków programowania, w tym języka Python daje możliwość szybkiego i łatwego zaimplementowania prostej aplikacji rozproszonej wykonującej założone zadania przez projektanta.

**Diagramy UML**

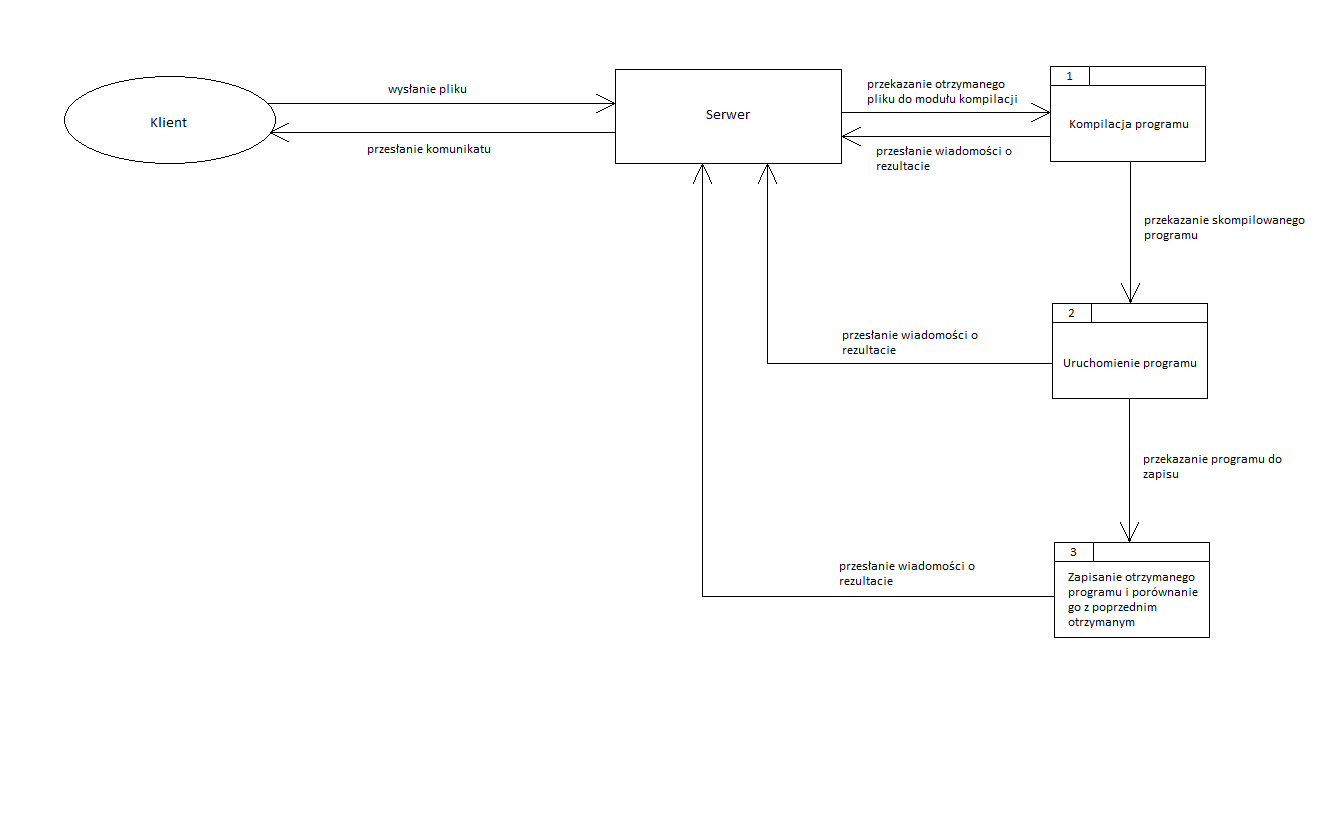
* **Diagram komunikacji**

****

* **Diagram sekwencji**

****

* **Diagram przepływu danych**

****

**Kody źródłowe oraz skrypty**

**Main.py**

**from** PythonREST **import** server, client, config  
**import** threading  
  
  
**class** Main:  
 k = client.Client  
 s = server.Server  
  
 **def** start(self):  
 print(**"starting program"**)  
 client = threading.Thread(target=self.k.run\_client, args=(self, 1), name=**"client"**)  
 server = threading.Thread(target=self.s.run, args=(self, config.IP, config.PORT), name=**"server"**)  
 client2 = threading.Thread(target=self.k.run\_client, args=(self, 2), name=**"client2"**)  
  
 client.start()  
 server.start()  
 client2.start()  
  
 server.join()  
 client.join()  
 client2.join()  
  
 **def** test\_all\_programs\_locally(self):  
 print(**"testing client-server based application:"  
 "\n1. creating server: IP={}, PORT={}"  
 "\n2. creating 5 client threads - each sending different program to server after few seconds delay:"  
 "\n3. printing communication output"  
 "\n\nstarting application..."**.format(config.IP, config.PORT))  
  
 s1 = threading.Thread(target=self.s.run, args=(self, config.IP, config.PORT), name=**"server"**)  
 c1 = threading.Thread(target=self.k.run\_client, args=(self, 1), name=**"c1"**)  
 c2 = threading.Thread(target=self.k.run\_client, args=(self, 2), name=**"c2"**)  
 c3 = threading.Thread(target=self.k.run\_client, args=(self, 3), name=**"c3"**)  
 c4 = threading.Thread(target=self.k.run\_client, args=(self, 4), name=**"c4"**)  
 c5 = threading.Thread(target=self.k.run\_client, args=(self, 5), name=**"c5"**)  
  
 s1.start()  
 c1.start()  
 c2.start()  
 c3.start()  
 c4.start()  
 c5.start()  
  
 s1.join()  
 c1.join()  
 c2.join()  
 c3.join()  
 c4.join()  
 c5.join()  
  
mk = Main()  
*# mk.start()*mk.test\_all\_programs\_locally()

**server.py**

**from** http.server **import** HTTPServer, SimpleHTTPRequestHandler  
**from** socketserver **import** ThreadingMixIn  
  
**from** PythonREST **import** executor, compiler, reporter  
*# import executor, compiler, reporter  
  
  
# import executor, compiler  
# import threading***class** Server(ThreadingMixIn, HTTPServer):  
 **def** run(self, IP, PORT):  
 *"""  
 starts server at given host and port* **:param** *IP: host's IP to start server* **:param** *PORT: server's port to listen requests* **:return***: None  
 """* server = Server((IP, PORT), MyHandler)  
 print(**"server started."**)  
 server.serve\_forever()  
  
  
**class** MyHandler(SimpleHTTPRequestHandler):  
 **def** \_set\_headers(self):  
 *"""sets headers"""* self.send\_response(200)  
 self.send\_header(**'content-type'**, **'text/html'**)  
 self.end\_headers()  
  
 **def** do\_PUT(self):  
 *"""  
 handles PUT requests. reads received file, tries to compile it and gives response to client about compilation.  
 if compilation was successful executes received program and sends response to client with received output.* **:return***: None - sends responds to client  
 """* length = int(self.headers[**'Content-Length'**])  
 content = self.rfile.read(length)  
 self.\_set\_headers()  
 self.wfile.write(**"file successfully received\n"**.encode())  
  
 **try**:  
 c = compiler.Compiler()  
  
 c.compile\_file(content)  
 self.wfile.write(**"compilation successful\n"**.encode())  
 **except** Exception **as** e:  
 self.wfile.write(**"compilation failed. Reason:"**.\_\_add\_\_(e.args).encode())  
  
 exec = executor.Executor()  
 test, result = exec.execute\_program(content)  
  
 **if** test:  
 self.wfile.write(**"program has been executed successfully and returned: "**.\_\_add\_\_(str(result)).encode())  
 **else**:  
 self.wfile.write(**"program execution failed "**.\_\_add\_\_(result).encode())  
  
 r = reporter.Reporter()  
 name = r.save\_program(content)  
 result = r.compare(name)  
 self.wfile.write(**"\nfiles with same content actually saved on server:\n"**.\_\_add\_\_(**"\n"**.join(result)).encode())  
 **if not** result:  
 self.wfile.write(**"None"**.encode())

Klasa Server pozwala na uruchomienie serwera na danym komputerze, podając jego adres IP oraz port, jeśli wywołana zostanie metoda Server((IP, PORT), MyHandler)gdzie argumentami jest adres IP, port oraz klasa MyHandler.

Klasa MyHandler odpowiedzialna jest za komunikację z połączonym klientem i odebranie żądania od niego. Metoda \_set\_headers ustawia nagłówki odpowiedzi na żądanie. Metoda do\_PUT odpowiada za:

* Przygotowanie i wysłanie komunikatu zwrotnego do klienta o odebranym pliku,
* Wywołanie metody compile\_filez klasy compiler, odpowiedzialnej za kompilację otrzymanego programu i wysłanie komunikatu o powodzeniu lub niepowodzeniu operacji do klienta,
* Wywołanie metody execute\_program z klasy executor odpowiedzialnej za uruchomienie skompilowanego programu i powiadomienie klienta o wyniku,
* Zapisanie i porównanie otrzymanego od klienta programu za pomocą metod save\_program oraz compare i powiadomienie klienta o wyniku operacji.

**client.py**

**import** http.client  
**import** os  
**import** time  
**from** PythonREST **import** config  
  
  
**class** Client:  
 path = os.getcwd().\_\_add\_\_(**"\Programs\\"**)  
 file1\_path = **"{}{}"**.format(path, **"anotherLongTimeProgram.py"**)  
 file2\_path = **"{}{}"**.format(path, **"example.py"**)  
 file3\_path = **"{}{}"**.format(path, **"fail.py"**)  
 file4\_path = **"{}{}"**.format(path, **"longTimeProgram.py"**)  
 file5\_path = **"{}{}"**.format(path, **"sum.py"**)  
  
 @staticmethod  
 **def** load\_file(self, file\_path):  
 *"""  
 loads file content from given path"* **:param** *file\_path: filepath to file* **:return***: file content  
 """* file = open(file\_path)  
 **return** file.read()  
  
 @staticmethod  
 **def** send\_file(self, name, content, server\_ip, PORT):  
 *"""  
 sends content to server defined in client module* **:param***:  
 name: name of file to send  
 content: content of file to send  
 server\_ip: ip of server where content should be sent  
 PORT: port where server is listening for requests* **:return***: server response  
 """* conn = http.client.HTTPConnection(server\_ip, PORT)  
 conn.request(**"PUT"**, **"/"**.\_\_add\_\_(name), content)  
 **return** conn.getresponse()  
  
 **def** run\_client(self, id):  
 *"""  
 fakes client behavior with following steps:  
 1. loads file from built-in filepath based on received id  
 2. sends content of loaded file to server defined by IP and PORT variables set in configuration* **:return***: None - prints received responses  
 """* time.sleep(2)  
 print(**"client {} started."**.format(id))  
 print(**"client {} loading file..."**.format(id))  
 **if** id == 1:  
 content = Client.load\_file(self, file\_path=Client.file1\_path)  
 **elif** id == 2:  
 content = Client.load\_file(self, file\_path=Client.file2\_path)  
 **elif** id == 3:  
 content = Client.load\_file(self, file\_path=Client.file3\_path)  
 **elif** id == 4:  
 content = Client.load\_file(self, file\_path=Client.file4\_path)  
 **else**:  
 content = Client.load\_file(self, file\_path=Client.file5\_path)  
 print(**"file loaded. sending file number {}..."**.format(id))  
 response = Client.send\_file(self, str(id), content, config.IP, config.PORT)  
 print(response.status, response.reason, response.read().decode())

Klasa Client składa się z:

* Atrybutów
  + IP – przyjmuje adres IP serwera z pliku config.py
  + PORT – przyjmuje wartość portu serwera z pliku config.py
  + file\_path – przechowuje ścieżkę do pliku, który ma zostać przesłany na serwer
  + file\_path2 – przechowuje ścieżkę do pliku, który ma zostać przesłany na serwer
* Metody
  + load\_file(self, file\_path)- metoda odpowiedzialna za wczytanie pliku z podanej ścieżki
  + send\_file(self, name, content, server\_ip, PORT) – metoda tworząca połączenie między klientem, a serwerem
  + run\_client(self, id)- metoda zarządzająca wysyłaniem plików

**compiler.py**

**import** py\_compile  
  
**class** Compiler:  
  
 **def** compile\_file(self, content):  
 *"""  
 tries to compile file content* **:param** *content:* **:return***:  
 compilation success: 1  
 compilation error: 0  
 """* **try**:  
 py\_compile.compile(content)  
 **return** 1, **"success"  
 except** Exception **as** e:  
 **return** 0, e.args

Klasa Compiler odpowiedzialna jest za kompilację otrzymanego pliku. Metoda compile\_file wykonuje próbę kompilacji. Gdy zakończy się powodzeniem zwraca 1, jeżeli nie zwraca wyjątek.

**executor.py**

**class** Executor:  
  
 **def** execute\_program(self, content):  
 *"""  
 executes given content of program.* **:param** *content: content of program to execute* **:return***: output of executed program  
 """* result = dict()  
 print(**"executing file..."**)  
 **try**:  
 exec(content, dict(), result)  
 print(**"file executed."**)  
 **return** 1, result[**"result"**]  
 **except** Exception **as** e:  
 **return** 0, str(e.args)

Funkcja odpowiedzialna za uruchomienie skompilowanego programu.

**config.py**

PORT = 8080  
IP = **"192.168.0.3"**

Plik zawierający adres IP serwera oraz jego port.

**reporter.py**

**import** os  
**from** builtins **import** staticmethod  
**import** filecmp  
  
  
**class** Reporter:  
  
 path = **"/server/programs/"  
  
 def** save\_program(self, content):  
 *"""  
 creates new file with a given content at server's path containing programs* **:param** *content: program's code* **:return***: name of saved file  
 """* self.create\_file\_directory\_if\_not\_exists(self, self.path)  
  
 files = len(os.listdir(self.path))  
 name = **"program"**.\_\_add\_\_(str(files+1).\_\_add\_\_(**".py"**))  
 file = open(self.path.\_\_add\_\_(name), **'w'**)  
 file.write(content.decode())  
 file.close()  
 **return** name  
  
 **def** compare(self, name):  
 *"""  
 compares content of file with given name from built-in directory to another file contents in this directory* **:param** *name: name of file to compare* **:return***: list of file names with same content  
 """* result = []  
 **for** i **in** (os.listdir(self.path)):  
 **if** i != name:  
 comparison = filecmp.cmp(self.path.\_\_add\_\_(i), self.path.\_\_add\_\_(name), shallow=**False**)  
 **if** comparison:  
 result.append(i)  
 **return** result  
  
 @staticmethod  
 **def** get\_file\_path(self):  
 **return** self.path  
  
 @staticmethod  
 **def** create\_file\_directory\_if\_not\_exists(self, path):  
 **if not** os.path.exists(path):  
 os.makedirs(path)  
  
 @staticmethod  
 **def** does\_directory\_exist(self, path):  
 **return** os.path.exists(path)

Klasa Reporter zawiera atrybut path posiadający ścieżkę zapisu wyniku wykonanych operacji. Metoda save\_program(self, content) zapisuje zawartość otrzymanego pliku na serwerze. Kolejna metoda compare(self, name) porównuje zapisane programy i zwraca informację czy są podobne.

Metody get\_file\_path(self),create\_file\_directory\_if\_not\_exists(self, path) oraz does\_directory\_exist(self, path)są odpowiedzialne za zarządzanie utworzeniem katalogu w którym zostaną umieszczone wyniki porównań.

**Programy przesyłane do serwera**

**example.py**

**def** f():  
 **return "Hello!"**result = f()

Funkcja f() wywołana zwraca napis „Hello!”.

**fail.py**

prin(**"this fails"**)

Wypisuje napis „this fails”.

**longTimeProgram.py**

**def** f():  
 **import** time  
 **for** i **in** range(0, 5):  
 time.sleep(3)  
 **return "long time program finished after 15 seconds."**result = f()

Program pozwalający na mierzenie czasu. W naszym przypadku zatrzymuje czas na 3 s, pięciokrotnie zwracając po zakończeniu pętli „long time program finished after 15 seconds.”

**sum.py**

**def** sum(a, b):  
 **return** a+b  
  
result = sum(1, 2)

Program składający się z funkcji pozwalającej na sumowanie dwóch liczb.

**Sposób uruchomienia projektu**

1. Instalacja środowiska IDE – PyCharm
2. Instalacja interpretera języka python
3. Podanie adresu serwera i portu w pliku config.py
4. Podanie ścieżki do plików, które mają zostać wysłane w module clineta

file\_path = **"D:\\longTimeProgram1.py"**file\_path2 = **"D:\\longTimeProgram2.py"**

1. Uruchomienie serwera za pomocą wiersza poleceń Windows:

python server.py

1. Uruchomienie serwera za pomocą wiersza poleceń Windows:

python client.py

**Wymagania do poprawnego działania aplikacji**

main.py:

- poprawna implementacja serwera

- poprawna implementacja klienta

-dostarczenie konfiguracji

reporter.py

1. Save\_program – brak wymagań
2. Compare – podanie nazwy pliku do porównania

server.py

1. Run – podanie IP, portu oraz stworzenie klasy handlera do obsługi zapytań
2. Do\_put – wysłanie od klienta zapytania typu PUT

client.py

1. Load\_file – podanie ścieżki do pliku
2. Send\_file – podanie dowolnej nazwy pliku, zawartości tego pliku, adresu IP serwera oraz portu
3. Run\_client – podanie id, służącego do określenia który program ma zostać wysłany

compiler.py

1. Compile\_file – dostarczenie zawartości pliku

config.py

Poprawne uzupełnienie IP oraz portu

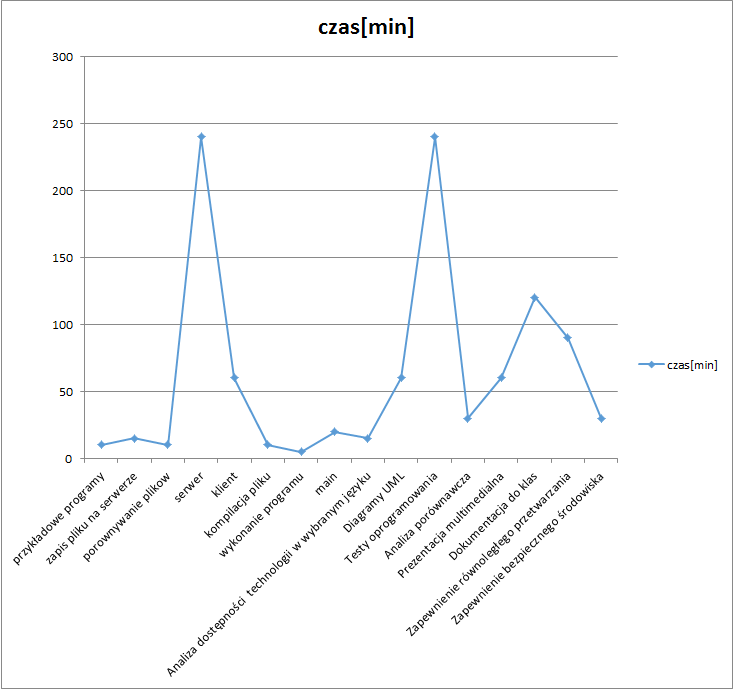
Executor.py

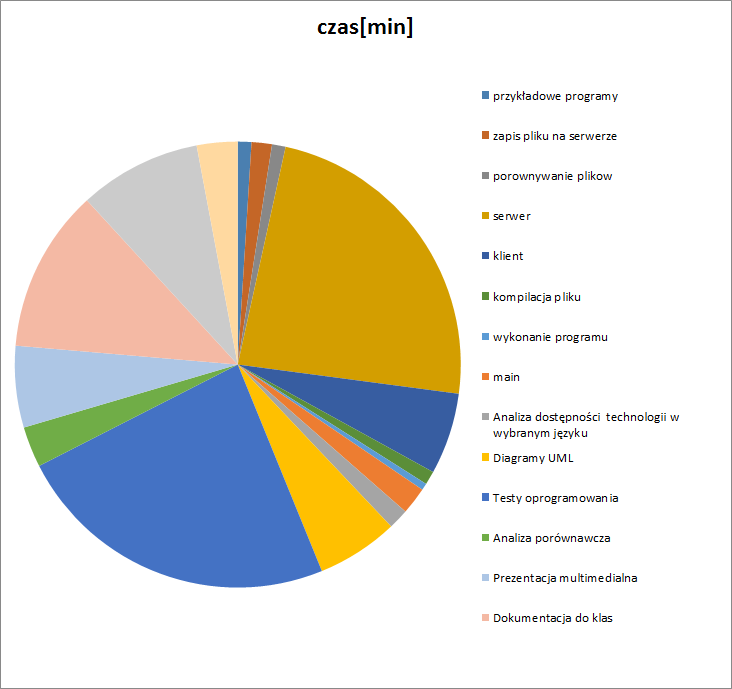
1. Execute program – dostarczenie zawartości pliku programu do wykonania, w którym do zmiennej result zostanie przypisana wartość zwracana przez program.

**Otrzymany wynik działania aplikacji**

testing client-server based application:  
1. creating server: IP=192.168.0.3, PORT=8080  
2. creating 5 client threads - each sending different program to server after few seconds delay:  
3. printing communication output  
  
starting application...  
server started.  
client 1 started.  
client 1 loading file...  
client 4 started.  
client 3 started.  
client 2 started.  
client 2 loading file...  
client 4 loading file...  
client 3 loading file...  
client 5 started.  
client 5 loading file...  
file loaded. sending file number 1...  
file loaded. sending file number 2...  
executing file...  
192.168.0.3 - - [15/Jun/2017 18:35:37] "PUT /1 HTTP/1.1" 200 -  
192.168.0.3 - - [15/Jun/2017 18:35:37] "PUT /2 HTTP/1.1" 200 -  
executing file...  
file executed.  
file loaded. sending file number 3...  
192.168.0.3 - - [15/Jun/2017 18:35:37] "PUT /3 HTTP/1.1" 200 -  
192.168.0.3 - - [15/Jun/2017 18:35:37] "PUT /4 HTTP/1.1" 200 -  
executing file...  
file loaded. sending file number 4...  
file loaded. sending file number 5...  
executing file...  
192.168.0.3 - - [15/Jun/2017 18:35:37] "PUT /5 HTTP/1.1" 200 -  
executing file...  
file executed.  
200 OK file successfully received  
compilation successful  
program has been executed successfully and returned: Hello!  
files with same content actually saved on server:  
None  
200 OK file successfully received  
compilation successful  
program execution failed ("name 'prin' is not defined",)  
files with same content actually saved on server:  
None  
200 OK file successfully received  
compilation successful  
program has been executed successfully and returned: 3  
files with same content actually saved on server:  
None  
file executed.  
200 OK file successfully received  
compilation successful  
program has been executed successfully and returned: long time program finished after 12 seconds.  
files with same content actually saved on server:  
None  
file executed.  
200 OK file successfully received  
compilation successful  
program has been executed successfully and returned: long time program finished after 15 seconds.  
files with same content actually saved on server:  
None

**Analiza porównawcza implementacji**

****

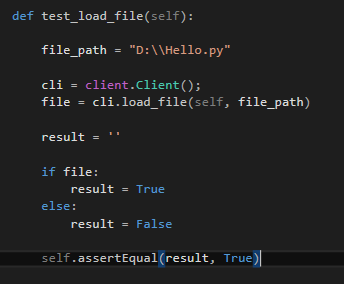
****

**Testy jednostkowe**

Do kodu źródłowego programu zostały napisane testy jednostkowe. Przy tworzeniu współczesnych systemów duży nacisk kładzie się na bezpieczeństwo. Ważne jest, aby aplikacje miały jak najmniej błędów, o które bardzo łatwo przy modyfikacjach kodu źródłowego. Testy definiują zachowania metod i klas. Istnieje także wzorzec TDD, którego założeniem jest najpierw pisanie testów do jeszcze nie istniejącego kodu, dzięki czemu powstałe metody mają określone jasno sprecyzowane cele.   
Testy jednostkowe pozwalają na wykonanie operacji testujących kod źródłowy. Dzięki temu programista przed udostępnieniem swojego kodu do repozytorium może sprawdzić poprzez uruchomienie testów czy jego zmiany nie zaburzyły działania całej aplikacji.  
W aplikacji pythonREST metody klas zostały przetestowane przy użyciu biblioteki unittest, która pozwala na sprawdzenie działania metody poprzez wywołanie jej a następnie porównanie wyniku metody z wartością oczekiwaną.

**Przykładowy test jednostkowy:**

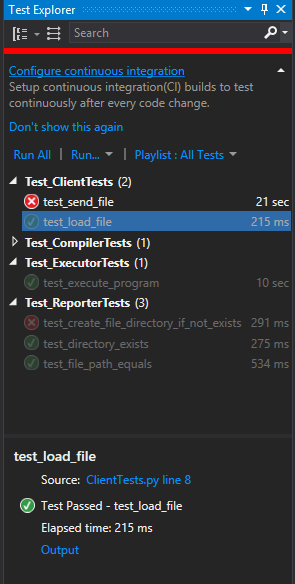
Test klasy Client.py



Testowi została poddana metoda load\_file klasy Client. Zadaniem metody jest załadowanie pliku z dysku.  
  
Biblioteka unittest udostępnia wiele metod pozwalających na testowanie zachowania. Poniższa metoda assertEqual pozwala sprawdzić czy wynik jaki otrzymamy po wykonaniu metody load\_file() jest zgodny z tym co zakłada programista

W zależności od IDE możliwości uruchomienia testów jednostkowych jest kilka. Jednakże sprowadzają się one do narzędzia udostępnionego w IDE ( w przypadku VisualStudio jest to TestExplorer ) i oczywiście zainstalowanych odpowiednich bibliotek umożliwiających programowanie w środowisku języka PYTHON.

**TestExplorer**

****Narzędzie TestExplorer pozwala na wykonanie testów jednostkowych. Można wykonać wszystkie jednocześnie lub pojedynczo testy, które nas interesują.

Jeżeli test nie przejdzie, zostanie oznaczony czerwonym krzyżykiem, a w informacji output poznamy przyczynę.

Test jednostkowe można debuggować tak jak zwykły kod. Jednakże żeby to zrobić konieczne jest wykonanie testu ręcznie – bez narzędzia Test Explorer.

**Bibliografia**

UML:

<http://zasoby.open.agh.edu.pl/~09sbfraczek/> (Beata Frączek, 14.06.2017)

REST

<https://docs.python.org/3/library/internet.html> (15.06.2017, Python Software Foundation)

<https://docs.python.org/3/library/http.server.html> (15.06.2017, Python Software Foundation)

<https://docs.python.org/3/library/http.client.html> (15.06.2017, Python Software Foundation)

<http://www.yarpo.pl/2012/07/29/rest-ciekawszy-sposob-na-komunikacje-client-server/> (29.07.2012)

<https://www.fullstackpython.com/api-creation.html> ([Matt Makai](https://github.com/mattmakai), [2012-2017](https://www.fullstackpython.com/change-log.html))

<https://techietweak.wordpress.com/2015/03/30/http-restful-api-with-python-requests-library/> (30.03.2015)

<https://developers.google.com/drive/v3/web/quickstart/python> (09.03.2017)

<http://documentation.commvault.com/commvault/v10/article?p=features/rest_api/rest_api_getting_started_python.htm> (13.06.2017)

UNIT TESTS

<https://martinfowler.com/bliki/UnitTest.html> ([Martin Fowler](https://martinfowler.com/), 05.05.2014)

<https://martinfowler.com/articles/mocksArentStubs.html> ([Martin Fowler](https://martinfowler.com/), 02.01.2017)

<https://docs.python.org/3/library/unittest.html> (15.06.2017, Python Software Foundation)

<https://developer.salesforce.com/page/How_to_Write_Good_Unit_Tests> (01.12.2014)