

UNIVERSITATEA TEHNICĂ „GHEORGHE ASACHI” IAȘI  
FACULTATEA AUTOMATICĂ ȘI CALCULATOARE  
SPECIALIZAREA TEHNOLOGIA INFORMAȚIEI

# EVALUAREA PERFORMANȚELOR

Aplicație pentru vizualizarea vremii

**Studenti:**

Strilciuc Gabriel

Răschitor G.D. Georgiana

Panainte I. Ancuța

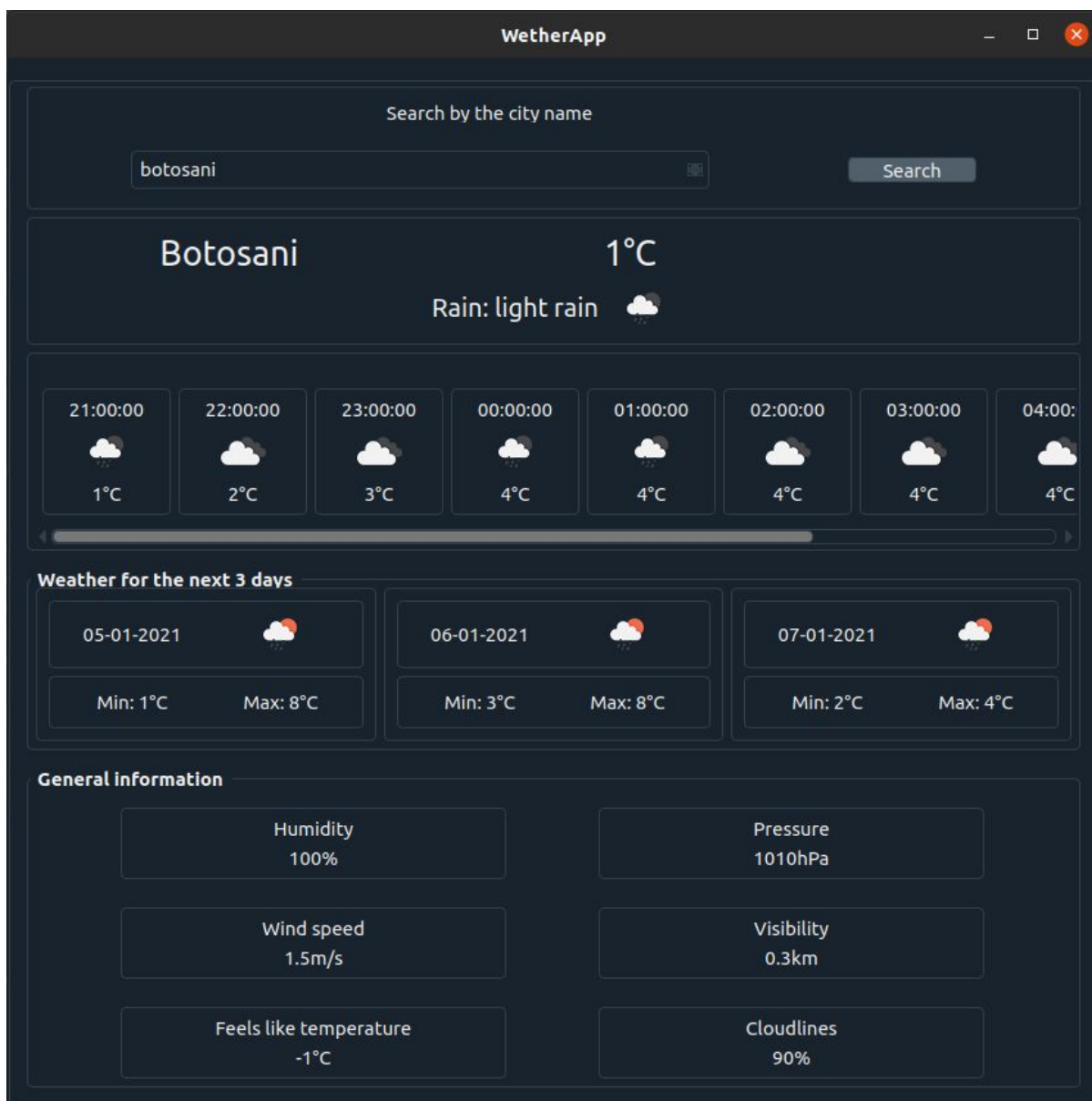
2020-2021

# Cuprins

1. Descriere proiect	3
2. Arhitectura generală	4
3. Funcționalitate	5
4. Complexitate	7
5. Testare	10

# Descriere proiect

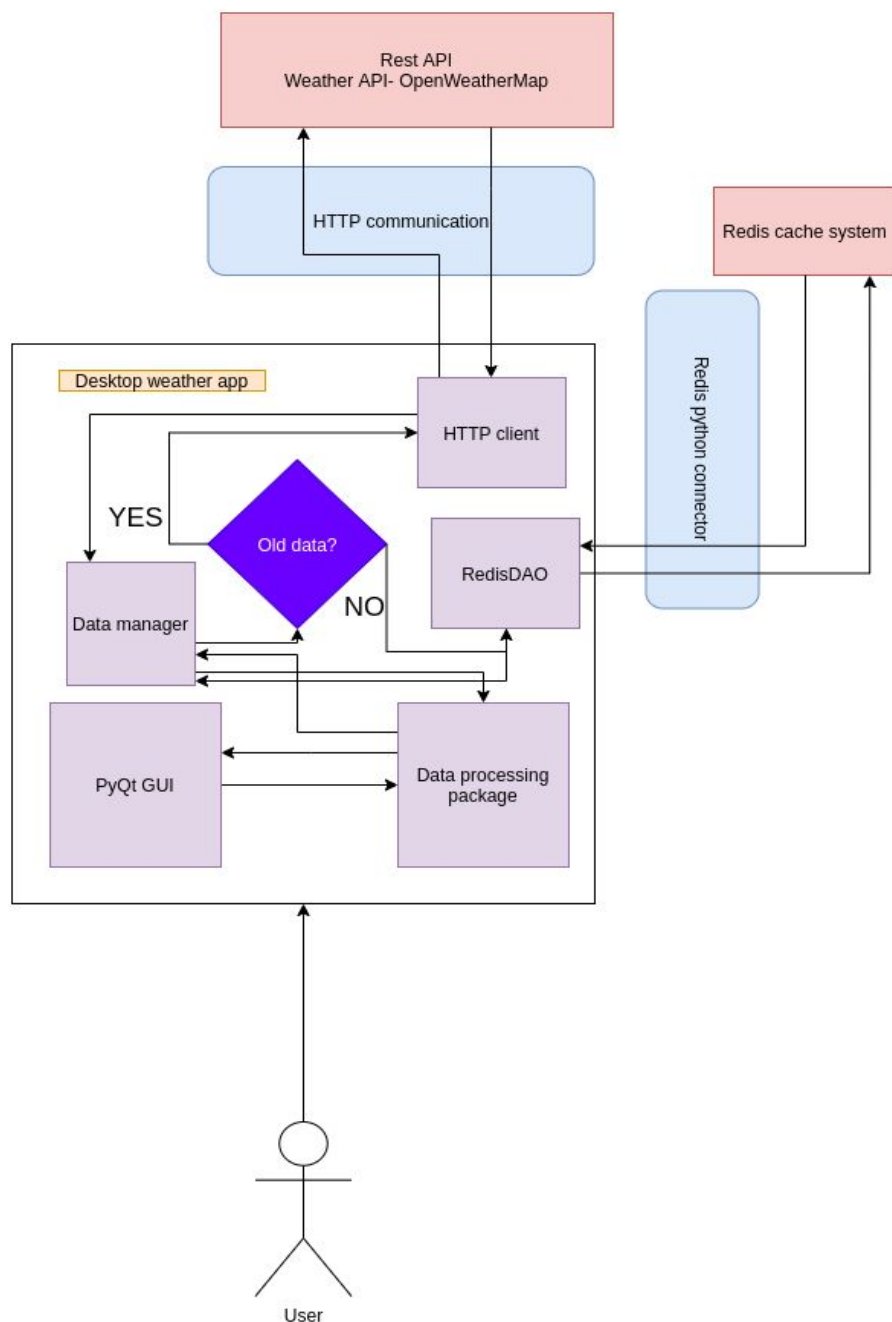
Acest proiect constituie o aplicatie pentru vizualizarea vremii, avand dat un nume de oras, in limba engleza. S-a realizat o interfata grafica cu utilizatorul pentru o mai buna vizualizare si folosire a aplicatiei. Interfata grafica a fost realizata cu ajutorul librariiei PyQt5 si Qt Designer. Algoritmul din spate implementand un un simplu fetch de date de la Rest API corespunzator si o procesare simpla de date.



# Arhitectura generala

Arhitectura proiectului consta in stack-ul de tehnologie: Qt, Redis si Python3. Acestea puse cap la cap realizeaza o aplicatie de tip desktop ce isi ia datele de pe Api-ul OpenWeatherMap. Poate fi pusa si problema unei lipse de chei pentru Api valabile, dar fiind o aplicatie ce nu va fi lansata pe internet, problema nu este aprofundata. Aceasta aplicatie reprezinta un exercitiu.

## Diagrama arhitecturii



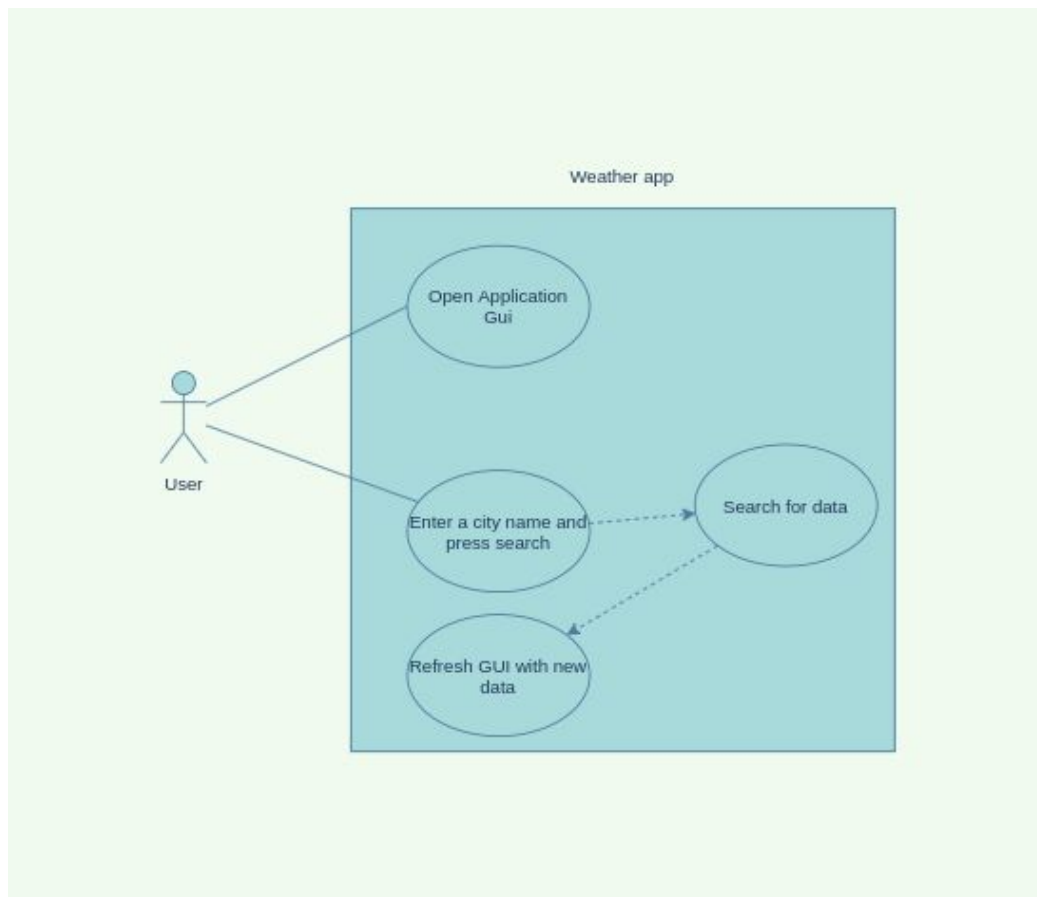
# Functionalitate

Aplicatia dispune si de un sistem de caching realizat cu middleware-ul Redis: o baza de date in-memory, fara persistenta a datelor, care salveaza un set de cheie:valoare in baza de date. Pentru sistemul de caching, logica de functionare se bazeaza pe faptul ca datele sunt valabile doar pentru o perioada scurta, apoi ele se sterg.

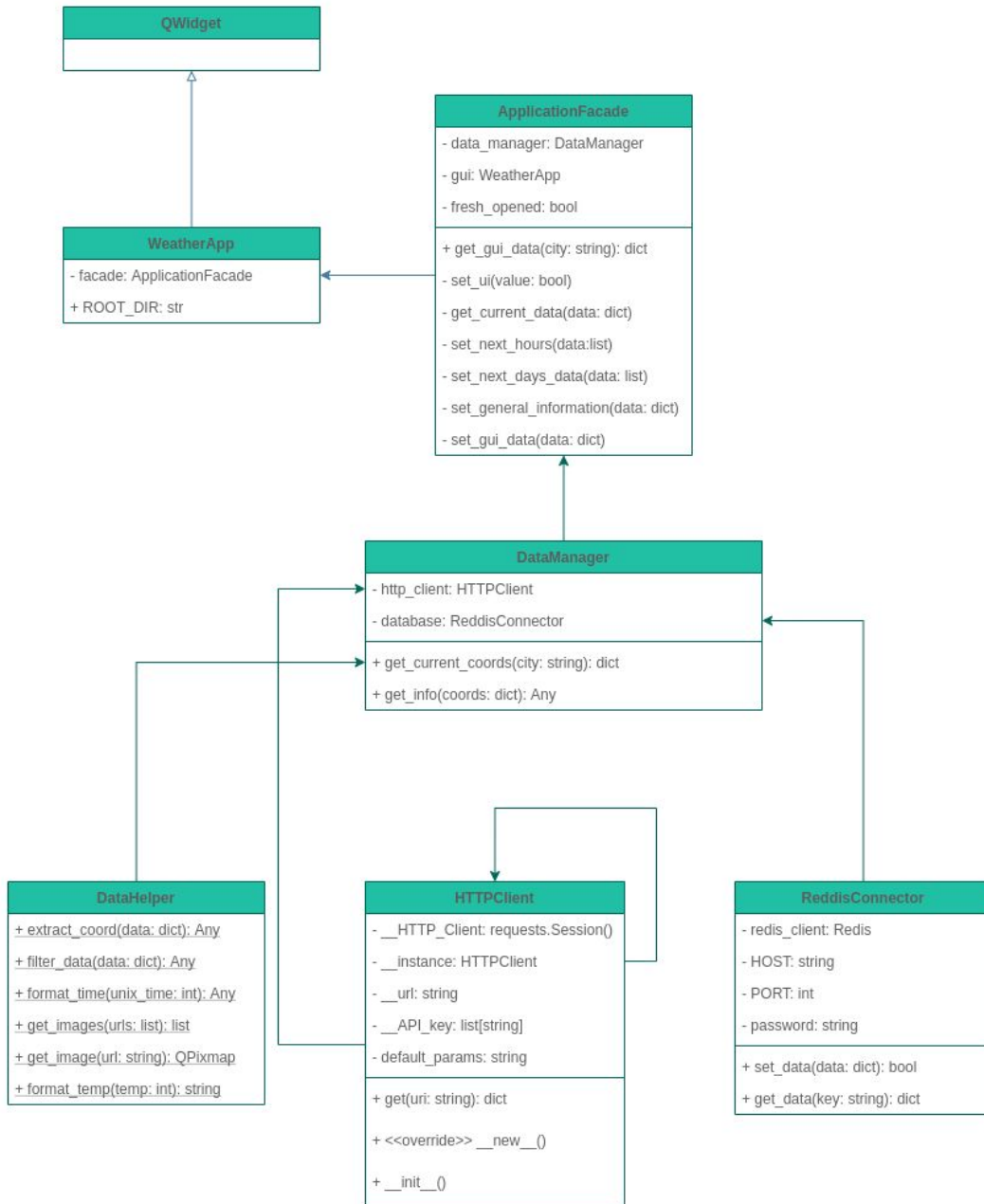
Datele stocate in baza de date Redis sunt valabile pana la urmatoarea ora fixa: de exemplu daca sunt introduse un set de date noi la ora 13:03, ele vor fi valabile pana la ora 14:00. In cele din urma dupa ce datele expira acestea sunt sterse automat de care Redis.

Datele care sunt aduse fie din sistemul de baze de date Redis fie cu cate un request HTTP de la REST API, sufera modificari de formatare pentru a avea o reprezentare cat mai clara in aplicatie.

## Diagrama de utilizare



# Diagrama de clase



# Complexitate

Complexitatea aplicatiei consta in modul de procesare a datelor primite de la API sau din baza de date. Astfel punctele de interes sunt reprezentate de clasele: ApplicationFacade, DataHelper si HTTPClient.

In clasa HTTPClient se itereaza cheile de autentificare catre API, astfel se intocmeste o complexitate  $O(n)$  unde  $n$  reprezinta numarul de chei alocate pentru aplicatie. Aceasta reprezinta complexitatea functiei `get()`.

```
try:
    keys = iter(self.api_keys)
    key = next(keys, None)

    while True:
        request_url = f"/data/2.5/{url}&appid={key}"
        res = self.connection.get(self.api_url + request_url)
```

In clasa DataHelper functiile de interes sunt: `get_images(urls)` si `filter_data(data)`. Prima functie realizeaza apeluri HTTP asincron pentru o lista de url-uri si returneaza o lista de imagini. Se realizeaza 3 iteratii: una care lanseaza threaduri in executie cu requesturi catre url-ul corespunzator, alta care colecteaza rezultatele si a treia functie ce transforma raspunsurile HTTP in imagini. Ca rezultat obtinem o complexitate timp de  $O(n)$  unde  $n$  reprezinta numarul de imagini, respectiv numarul de url-uri pentru care se fac cererile. Cea de a doua functie itereaza o lista de obiecte cu date zilnice si o lista de obiecte cu date pe ora. Fiindca sunt considerate tot timpul primele 10 ore si primele 3 zile complexitatea timp este una constanta  $\Rightarrow O(1)$ .

```
# Extract data for daily representation
for daily_item in daily[:3]:
    tmp_min_temp = daily_item["temp"]["min"]
    tmp_max_temp = daily_item["temp"]["max"]
    tmp_icon_uri = "http://openweathermap.org/img/wn/{0}.png".format(daily_item["weather"][0]["icon"])
    tmp_time = FormatDataHelper.format_time(daily_item["dt"])
    tmp_description = daily_item["weather"][0]["main"]

    daily_item = {
        "time": tmp_time,
        "icon": tmp_icon_uri,
        "min_temp": round(tmp_min_temp),
        "max_temp": round(tmp_max_temp),
        "description": tmp_description
    }
```

```

with futures.ThreadPoolExecutor(max_workers=10) as e:
    fs = [
        e.submit(lambda: requests.get(url)) for url in urls
    ]
    results = [
        f.result() for f in fs
    ]

    def build_pixmap(response):
        if response.status_code == 200:
            image_data = response.content
            pixmap = QPixmap()
            pixmap.loadFromData(image_data)
            return pixmap
    return list(map(lambda x: build_pixmap(x), results))

```

In clasa ApplicationFacade, se realizeaza managementul interfetei, iar functiile ce reprezinta o complexitate indusa de aplicatia dezvoltata sunt: `set_next_hours(data)`, `set_next_days_data(data)` si `set_general_information(data)`. Prima functie introduce datele obtinute si filtrate de aplicatie in interfata grafica. Deoarece numarul de ore este tot timpul 10, putem considera complexitatea timp una  $O(1)$ . Cea de-a doua functie realizeaza doua iteratii, una pentru obtinerea imaginilor necesare si una pentru a introduce datele. Dat fiind faptul ca sunt reprezentate doar 3 zile, din nou complexitatea timp este considerata constanta,  $O(1)$ . Ultima functie acceseaza etichetele informatiilor generale curente. Se itereaza un numar de 6 chei de dictionar si din nou actiunea se realizeaza in timp constant  $O(1)$ .

```

for key in label_names.keys():
    label = self.__gui.findChild(QLabel, label_names[key])
    value = data[key]
    label.setText(str(value))

```



```
icons_url = list(map(lambda x: x["icon"], data))
results = FormatDataHelper.get_images(icons_url)

for index, item in enumerate(data):
    # Search for the labels to be changed
    label_img = self.__gui.findChild(QLabel, tmp_img+str(index+1))
    label_time = self.__gui.findChild(QLabel, tmp_time+str(index+1))
    label_temp = self.__gui.findChild(QLabel, tmp_temp+str(index+1))

    # Update the labels
    label_time.setText(str(item["time"].split("~")[0].strip()))
    label_temp.setText(str(FormatDataHelper.format_temp(item["temperature"])))
    label_img.setPixmap(results[index])
```

# Testare

Pentru testarea aplicatiei s-a utilizat testarea unitara, prin care a fost asigurata functionarea corecta a fiecarei clasei.

Aceasta a fost realizata folosind modulul **unittest** din python.

**DataManagerTest** – testarea corectitudinii datelor primite in urma cererilor pentru anumite orase sau coordonate. Daca orasul introdus este valid se va returna un json cu datele relevante, iar in caz contrar se asteapta o exceptie cu mesajul “city not found”.

```
# test the city returned is the right city
def test_city(self):
    data_manager = DataManager()
    city = data_manager.get_info("Amsterdam")
    with open('amsterdam.txt', 'r') as file:
        f = file.read()
    city_expected = json.loads(f)
    self.assertEqual(city["city"], city_expected["city"])
```

```
# test message for non-existent city
def test_city_not_found(self):
    data_manager = DataManager()
    with self.assertRaises(Exception):
        city = data_manager.get_info("vazlui")
```

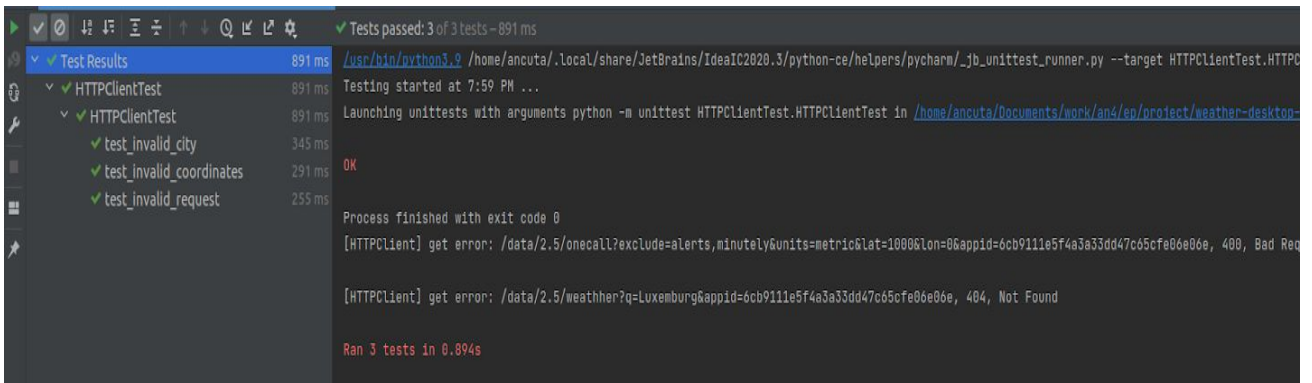
```
✓ Tests passed: 5 of 5 tests - 1 s 240 ms
✓ Test Results 1 s 240 ms /usr/bin/python3.9 /home/ancuta/.local/share/JetBrains/IdeaIC2020.3/python-ce/helpers/pycharm/_jb_unittest_runner.py
  ✓ DataManagerTest 1 s 240 ms Testing started at 8:47 PM ...
    ✓ DataManagerTest 1 s 240 ms Launching unittests with arguments python -m unittest DataManagerTest.DataManagerTest in /home/ancuta/Documents/work/a
      ✓ test_city 3 ms
      ✓ test_city_not_found 376 ms
      ✓ test_coords_invalid_city 373 ms
      ✓ test_coords_valid_city 487 ms Ran 5 tests in 1.245s
      ✓ test_filtered_data 1 ms OK
      Process finished with exit code 0
      [DataManager] Take data with key='Amsterdam' from database
      [DataManager] New request with key='vazlui'
      [DataManager] Take data with key='Amsterdam' from database
```

## HTTPClientTest – verificare raspunsuri pentru diferite query-uri trimise catre API

De exemplu, pentru un oras invalid API-ul va returna un json prin care se va indica faptul ca orasul nu a fost gasit. La fel si pentru o cerere cu latitudinea si longitudinea gresite sau pentru orice alta eroare internă.

```
# test query with invalid city
def test_invalid_city(self):
    http_cli = HTTPClient()
    url = "weather?q=Luxemburggg"
    data = http_cli.get(url)
    exp_msg = {'message': 'city not found'}
    self.assertEqual(data, exp_msg)
```

```
# test query with invalid coordinates
def test_invalid_coordinates(self):
    http_cli = HTTPClient()
    lat = 1000
    lon = 0
    url = f"onecall?exclude=alerts,minutely&units=metric&lat={lat}&lon={lon}"
    data = http_cli.get(url)
    exp_msg = {'cod': '400', 'message': 'wrong latitude'}
    self.assertEqual(data, exp_msg)
```



```
✓ Tests passed: 3 of 3 tests - 891 ms
Test Results
  ✓ HTTPClientTest 891 ms /usr/bin/python3.9 /home/ancuta/.local/share/JetBrains/ideaIC2020.3/python-ce/helpers/pycharm/_jb_unittest_runner.py --target HTTPClientTest.HTTPC
    ✓ HTTPClientTest 891 ms Testing started at 7:59 PM ...
      ✓ test_invalid_city 345 ms Launching unittests with arguments python -m unittest HTTPClientTest.HTTPClientTest in /home/ancuta/Documents/work/ana4/ep/proiect/weather-desktop-
      ✓ test_invalid_coordinates 291 ms OK
      ✓ test_invalid_request 255 ms Process finished with exit code 0
[HTTPClient] get error: /data/2.5/onecall?exclude=alerts,minutely&units=metric&lat=1000&lon=0&appid=6cb9111e5f4a3a33dd47c65cfe06e06e, 400, Bad Req
[HTTPClient] get error: /data/2.5/weather?q=Luxemburg&appid=6cb9111e5f4a3a33dd47c65cfe06e06e, 404, Not Found
Ran 3 tests in 0.894s
```

**FormatDataHelperTest** – testarea datelor returnate in urma filtrarii si prelucrarii informatiilor primite de la clientul HTTP

```
# test extract_coord method
def test_extract_coord(self):
    format_data = FormatDataHelper()
    data = {"coord": {"lon": -0.13, "lat": 51.51}}
    expected_data = {"lon": -0.13, "lat": 51.51}

    result = format_data.extract_coord(data)
    self.assertEqual(result, expected_data)
```

```
# test format_time method
def test_format_time(self):
    format_data = FormatDataHelper()
    dt = 1609765942
    expected_data = "15:12:22 ~ 04-01-2021"
    result = format_data.format_time(dt)
    self.assertEqual(result, expected_data)
```



The screenshot shows the PyCharm test runner interface. At the top, it indicates 'Tests passed: 3 of 3 tests - 4 ms'. Below this, a tree view shows the test results for 'FormatDataHelperTest', which includes three sub-tests: 'test\_extract\_coord' (1 ms), 'test\_filter\_data' (3 ms), and 'test\_format\_time' (0 ms). All tests are marked as passed with green checkmarks. The bottom of the window shows a summary: 'Ran 3 tests in 0.006s', 'OK', and 'Process finished with exit code 0'.

**RedisConnectorTest** – verificare setare si preluare date din Redis  
Adaugarea unui dictionar de test

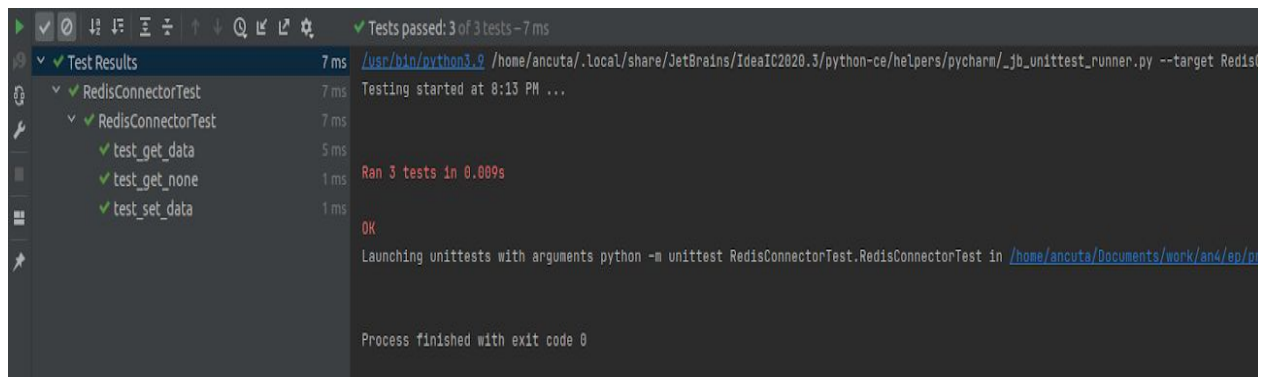
```
# test set data in Redis
def test_set_data(self):
    database = RedisConnector()
    key = 'test'
    data = {'data': 'test data'}
    exp_resp = database.set_data(key, data)
    self.assertEqual(True, exp_resp)
```

Preluarea lui din baza de date

```
# test get data from Redis
def test_get_data(self):
    database = RedisConnector()
    key = 'test'
    data = database.get_data(key)
    exp_data = {'data': 'test data'}
    self.assertEqual(data, exp_data)
```

Verificare rezultat returnat pentru date care nu exista

```
# test get nonexistent data from Redis
def test_get_none(self):
    database = RedisConnector()
    key = 'randomkey'
    data = database.get_data(key)
    self.assertEqual(data, None)
```



Sarcinile de lucru au fost distribuite astfel:

Strilciuc Gabriel

- implementare RedisConnector, ApplicationFacade
- interfata

Panainte Ancuța

- implementare DataManager, HTTPClient
- testare RedisConnector, HTTPClient

Rășchitor Georgiana

- implementare FormatDataHelper
- testare DataManager, FormatDataHelper
- documentatie