

# Tema 2: Servicii de vreme

- Servicii Web si Docker -

Vlad-Constantin Lungu-Stan 341C4

Asta a fost una dintre temele alea din facultate la care nu aveam habar mai deloc nimic la începutul ei(impropriu spus, am făcut laburile de docker si Rest, dar nu aveam o înțelegere complete asupra lor). Asta a însemnat că a trebuit să învăț două chestii noi, ceea ce a fost fun și interesant. Să începem...

## 1. Docker Compose

### 1.1 Baza de date

Primul pas în rezolvarea temei a fost alegerea unei baze de date. Cum experiența mea cu bazele de date se rezumă la un semestru cu un prof care nu-și ținea cursurile și ceva joacă în liceu în MySQL, am decis să ascult poporeni și să folosesc **Postgresql**. Un alt motiv pentru folosirea ăsteia a fost că în enunț era un exemplu cu docker compose cu postgresql, așa că mi-am zis "de ce nu???". Am căutat pe net, am citit niște tutoriale, am văzut și exemplul din enunțul temei și am scris partea de baze de date din docker compose: partea din services notată cu "db".

I-am dat un nume sugestiv containerului ca să îl pot referi mai ușor în relație cu celelalte containere, i-am setat imaginea care era și în exemplu, am pus restart always că așa am găsit pe net, după am trecut la partea de care aveam idee pentru că ne-a fost explicat bine la laborator: networks. Baza asta de date are conexiuni în două părți, pe de-o parte cu api-ul pe care îl construim noi, pe de altă parte cu utilitarul de gestiune a bazei de date. Pentru fiecare am făcut **o rețea separată**:

- **db\_network** pentru comunicarea între utilitar și baza de date.
- **api\_network** pentru comunicarea între api și baza de date.

Astfel, baza de date este singurul container care se află în două rețele diferite. Portul expus de container este și cel standard, anume **5432**. Desigur, i-am făcut și un volum în care să-și păstreze datele, numit db\_data. După aia am încercat să-l rulez și n-a vrut, spunând că are nevoie de user și parolă. Date fiind cele discutate, am adăugat o serie de variabile de mediu în secțiunea environment, anume **numele bazei de date**(Tema2), **Userul**(Tema2\_user), **PAROLA**(1234) și pgdata, unde-și ține datele(I guess). Știu că nu e bine să pui parola în clar, dar asta e o temă, e un exemplu didactic și iese în afara scopului acestei teme(și probabil a materiei), așa că nu mi-am bătut mai mult capul cu secrete/fișiere de parole sau alte nebunii.

Ulterior am făcut și **persistentă inițializarea bazei de date** ca să nu fii nevoit să creezi de mână baza de date la fiecare rulare. Cred că asta intră în categoria "optimizarea configurației de containere". Am făcut un fișier numit "init\_db.sql" care se află în initialization\_scripts și care conține codul bun de creare a tabelelor din baza de date, Tari, Orase și Temperaturi. Toate tabelele au câmpul id generat automat by default(ca să poți schimba eventual id-urile dacă vrei, deși nu era cerut în temă) și primary key. Tipurile sunt cele cerute în enunț. Tabela **Tari** are câmpul **nume** setat **UNIQUE**, așa cum a fost precizat în enunț. Tabela **Orașe** are câmpul **id\_tara** **FOREIGN KEY** cu id-ul din Tari cu **UPDATE/DELETE cascade**. Am adăugat o constrângere numită "unique\_tara\_oras" pentru a garanta **unicitatea** perechii de attribute (**id\_tara, nume\_oras**), după cum era specificat. Tabela **Temperaturi** are câmpul timestamp de tipul

Timestamp, adică reține pe lângă dată și ora, minutul, secunda(cred că și milisecunda). Am preferat acest format pentru că mi se pare normal ca un oraș să poată avea mai multe temperaturi identificate într-o singură zi. Idem câmpului id\_tara din tabela Orase, câmpul id\_oras este cheie străină cu id-ul de oraș cu **UPDATE/DELETE cascade**. De asemenea, am adăugat și constrângerea de **unicitate** pentru **(id\_oras,timestamp)** pentru că era specificată în enunț. Desigur, ținând atât de fin timpul, nu o să fie niciodată folosită, două timestampuri identice fiind foarte greu de obținut(nici nu poți face update la timestamp cu PUT).

Din câte am citit, la inițializare postgresql se uită în folderul docker-entrypoint-initdb.d și execută toate fișierele de inițializare, în ordine alfabetică, dacă rezultatul lor nu se găsește deja în baza de date. Așa că pentru asigurarea persistenței am folosit **fișierul de inițializare** proaspăt creat ca **volum** și l-am pus în folderul mai sus menționat. Această încercare a fost un real succes, la pornirea utilitarului **conținutul bazei de date fiind reprezentat de cele 3 tabele goale**.

## 1.2 Utilitarul de gestiune a bazei de date

Folosind Postgresql, alegerea evidentă a fost utilizarea **PgAdmin** ca utilitar de gestiune a bazei de date. Configurarea lui am făcut-o în partea de "sgbd" a docker-compose-ului. l-am dat un nume, imaginea, l-am băgat în aceeași rețea cu baza de date și am **expus serviciul pe PORTUL 8081**(alegere complet aleatorie, l-am mai văzut cred că în labul de docker folosit pe 8080, dar am vrut să-l păstrez pe ăla pentru api, așa că l-am pus pe următorul disponibil). De asemenea a trebuit să setez ca variabile de mediu credențialele utilizatorului default, care sunt **tema2@tema2.com** cu parola **1234** și portul pe care ascultă, 80.

Ca să asigur o inițializare corectă a containerelor, am pus o clauză depends\_on pentru sgbd cu baza de date. Astfel, mai întâi pornesc baza de date și după aceea se pornește pg\_admin.

Pentru accesarea utilitarului trebuie accesat localhost:8081, introdus numele de utilizator **(tema2@tema2.com)** și introdusă parola **1234**. În acest moment ne întâmpină o pagină goală, lucru care nu mi-a plăcut. Vreau ca atunci când pornesc utilitarul să-mi și văd serverul de baze de date acolo și să-l accesez. Așa că am săpat puțin prin documentație și am venit cu un json de configurare inițială a serverelor pe care-l folosesc ca volum pentru calea "/pgadmin4/servers.json", care reprezintă fișierul în care își salvează( din câte am înțeles) serverele. Acest fișier se află în folderul initialization\_scripts și conține datele necesare pentru adăugarea serverului: nume, grupul în care va fi adăugat, portul pe care răspunde, username-ul cu care să se conecteze și adresa de host. Aici a mers să-i dau numele containerului de baza de date, deci bănuiesc că **cerința de named DNS în cadrul aplicației pentru a putea referi containerele după nume și nu după adresa ip merge**.

După rularea docker volume prune și docker-compose -f .\tema\_2\_docker\_compose.yml up, am intrat în utilitarul de baze de date și serverul era acolo. Ce nu am reușit să fac să meargă a fost logarea automată la serverul respectiv. După ce am pierdut ceva vreme fără **să vrea să meargă**, am renunțat la treaba asta, considerând că nu e o muncă titanică să bagi parola **1234** când accesezi serverul.

Astfel partea de **optimizare a configurației de containere** cred că e satisfăcută. Am **variabile de mediu** la db și sgbd, am **două rețele distincte**, fiecare pentru componenta pe care o deservește( api + db, db+sgbd), am folosit **volum pentru inițializare și persistența datelor** și am dat **nume containerelor** ca să mă pot folosi de ele în adresare.

### 1.3 API

Containerul care reprezintă serverul este declarat în partea "api" a fișierului docker\_compose. Containerul se numește **rest\_server**. Spre deosebire de celelalte două containere care, evident, folosesc imagini de docker existente, aceasta **își construiește propria imagine** de docker, motiv pentru care am partea de build: "build: ".". Astfel, folosește **Dockerfile-ul** și **.dockerignore-ul** din folderul curent pentru a construi imaginea de server.

**Dockerfile-ul** conține 8 reguli pentru construirea imaginii serverului și este aproape identic celui din laborator, cu care se construia imaginea de server de python. De ce nu m-aș fi folosit de asta, până la urmă? Folosesc Python, așa că prima dată specific imaginea pe baza căruia se construiește imaginea curentă ca fiind **Python3.6**, cu clauza **FROM**. După aceea copiez **fișierul de requirments**, care conține dependențele serverului în folderul de temporare cu **COPY**. Urmează două reguli de **RUN** prin care sunt instalate cu pip prima dată setuptools(pentru management pachete), apoi dependențele din fișierul de requirments. **Dependențele** sunt minimale: **flask**(ca să funcționeze ca un server), **psycopg2**(ca să relaționez cu baza de date) și **datetime**(pentru timestampuri). După aceea **copiez folderul "Server"**, care nu conține altceva decât fișierul python care implementează serverul, **în /app** tot cu COPY, setez /app ca working directory cu WORKDIR, expun portul 80(era așa în laborator, am lăsat așa și eu) și **pornesc serverul** cu CMD, urmată de lista formată din programul de rulat, mai exact "python" și numele serverului, "server.py".

Fișierul **.dockerignore** este cu dute-vino. Am citit în documentație ce este, cum se construiește și ce ar trebui să facă și am acționat în consecință. Din păcate, însă, nu i-am văzut impactul. Un motiv pentru asta cred că e și faptul că toată tema mea are la momentul scrierii aproximativ 200KB. Oricum, am citit că e mai ușor decât să ștergi de mână ce nu-ți trebuie să **procedezi pe dos: ștergi tot și scrii reguli de excepție pentru ce ai nevoie**. Fix asta am făcut. Fișierul se parcurge secvențial, așa că prima regulă a fost **\*\*\***, care spune **"deny everything"**. După aceea am adăugat **3 reguli de excepție** pentru folderul **Server("!Server/")**, **conținutul folderului Server("!Server/\*")** și pentru **fișierul de dependențe ("!requirements.txt")**. Logica îmi spune că e bine, dar nu am găsit o modalitate bună de a verifica. Imaginea rezultată are vreo 900MB, însă cea mai mare parte e consumată de python, așa că voi considera că ce am făcut e bine. Sper că e ce trebuie!

Containerul serverului comunică doar cu serverul de baze de date prin intermediul rețelei api\_network. Acesta **expune portul 8080** și depinde, asemenea lui sgbd, de initializarea containerului de baze de date. După am văzut că nu merge partea de depends on și că în versiunea 3 a docker compose **depends\_on nu mai e suportat**, so tough luck :/. Eh, a fost o încercare nobilă.

### 1.4 Cum merge?

În fișierul Build.txt din folderul Readme se găsește outputul unei comenzi de docker compose up, după ce în prealabil am dat down, am dat volume prune și am dat system prune -a ca să mă asigur că totul este șters. Se creează networkurile și volumele, se downloadează imaginile pentru postgresql și pgadmin, apoi se construiește imaginea în 8 pași, aia 8 din dockerfile. Totul merge bine și după ce are imaginile creează cele 3 containere și le dă drumul. Prima dată pornește containerul de baze de date(probabil pentru că e primul în xml). Înainte să termine încearcă să pornească api-ul fără succes, pentru că serverul de postgresql nu a terminat inițializarea. Asta e dovada că depends\_on nu merge, sau merge parțial. După încă puțin se inițializează complet serverul de baze de date și reușește si api-ul să se

conecteze și să se pună pe picioare. După aceea, ce-i drept puțin mai greu, pornește și containerul de pgadmin, moment în care implementarea mea este funcțională!

Asta a fost partea de "Tehnologie Docker" a temei, cu imagini, networks, docker-compose, xml-uri, volume și tot tacâmul.

## 2 REST API

Cea de-a doua parte a temei și cea mai mâncătoare de timp este, desigur, Rest Api-ul construit de noi. Pentru implementarea lui am folosit singurul limbaj pe care îl știu care se pretează pentru așa ceva: **Python**. A fost suficient de versatil încât să-mi ofere tot ce am nevoie pentru realizarea temei. Pentru realizarea părții de server am folosit **Flask**, pentru că așa am făcut și la laborator și aveam exemplu. Pentru comunicarea cu serverul de baze de date am căutat ce s-ar preta cel mai bine și găsit biblioteca **psycopg2** care interfațează comunicarea cu servere de PostgreSQL. Yey, fix ce aveam nevoie! Desigur, am ignorat cu nonșalanță sfaturile de a folosi ceva numit sqlalchemy(zicea și termenul de ORM pe acolo) pentru că nu știu ce e aia și ar fi necesitat un learning curve pe care nu eram dispus să îl parcurg(a venit tema la SPG, avem teste de curs săptămânale la nu știu ce alte materii...e nasol anul asta 😞). În plus, știu să scriu cereri SQL și am considerat că am control mai mare asupra lucrurilor în caz de eroare dacă scriu cererile de mână și le trimit la server cu psycopg2. Am mai folosit și biblioteca **datetime** pentru a putea să ofer **timestampuri**.

Logica serverului e simplă și ușor de urmărit. La început import pachetele de care am nevoie, creez aplicația flask, mențin în variabile datele necesare pentru conectarea la serverul de baze de date și realizez conexiunea cu serverul de postgresql. În partea de jos am un main mai mult decât minimal, unde doar rulez aplicația pe localhost. Automat o pune pe portul 5000. Nu am făcut alte modificări la partea de comunicație de rețea pentru că nu am văzut rostul.

După asta, însă, vin vreo 400 de linii de cod pentru logica serverului. La început am 3 funcții ajutatoare. Prima, **check\_json** primește un json și o listă de tupluri (nume, tip) care verifică validitatea jsonului primit. la fiecare element din lista de elemente așteptate. Dacă vreunul nu se află în json sau are tipul diferit, întoarce False, marcând un json invalid. Aici intervine o problemă **în cazul tipului double**(float la mine, Python are float-ul double precision). Dacă primești un **int**(aka număr fără virgulă, aka 5 în loc de 5.0) și aștepti un **float**, verificarea **va considera că nu ai primit ce trebuie**. Asta e o problemă de tipare și nu foarte relevantă pentru scopul temei, însă aștept răspunsul pe forum ca să mă asigur că totul este în regulă. **PS: am văzut răspunsul, am realizat că trebuie să bag doar un if în plus așa că am adăugat și să meargă dacă primește int-uri(5) în loc de double(5.). Totuși, asta a fost adăugată ca ultimă schimbare, am testat-o puțin și merge, nu crapă testele deja scrise, dar nu pot să garantez că nu are absolut nicio problemă.** Momentan las tiparea așa cum e până la noi ordine. Cea de-a doua e **get\_record**, care primește la fel o listă de elemente și un json și extrage din json valorile elementelor din listă, pe care le pune într-o altă listă. Cea de-a treia, **make\_dict**, construiește un dicționar care va fi jsonificat, reprezentând o parte din răspunsul unei cereri **GET**. Aceasta ia o lista de tupluri(nume, tip) și o listă de valori și pune în dicționar, în dreptul numelui indicat, elementele din listă convertite la tipul corespunzător.

După aceea urmează 3 zone mari de implementare unde se găsesc path-urile la care trebuie să răspundă serverul. Fiecare zonă reprezintă cererile pentru una dintre tabele, ele fiind "Countries", "Cities" și "Temperatures". Fiecare zona are o structură identică. La începutul zonei îmi definesc listele

de tupluri (nume,tip) care vor fi utilizate în cererile căilor respective și string-uri care reprezintă cererile sql propriu-zise. După aceea trec la implementarea propriu zisă a cererilor, pe rând, așa cum era cerut în enunț. Implementarea e straight-forward, așa că nu o voi detalia.

Ce ar mai merita să fie menționat este cum rezolv partea de parametri lipsă din cererile de get pentru temperaturi. Am o suită de query-uri de bază pentru get\_orase, get\_tari, get\_all\_lat\_lon, get\_all\_lat, get\_all, care au intervalul de timp asociat "between a and b". Dacă e să lipsească latitudine, longitudinea sau amandouă schimb query-ul de bază. Dacă e să lipsească vreun capăt de interval am funcția modify\_time care transforma between a and b în timestamp >= x sau timestamp <= x.

Mai e de menționat ceva, totuși. Pentru cererile get de la temperatures, dacă unul dintre parametri este nevalid nu prea contează, cererea va da eroare la serverul de baze de date și voi întoarce lista vidă.

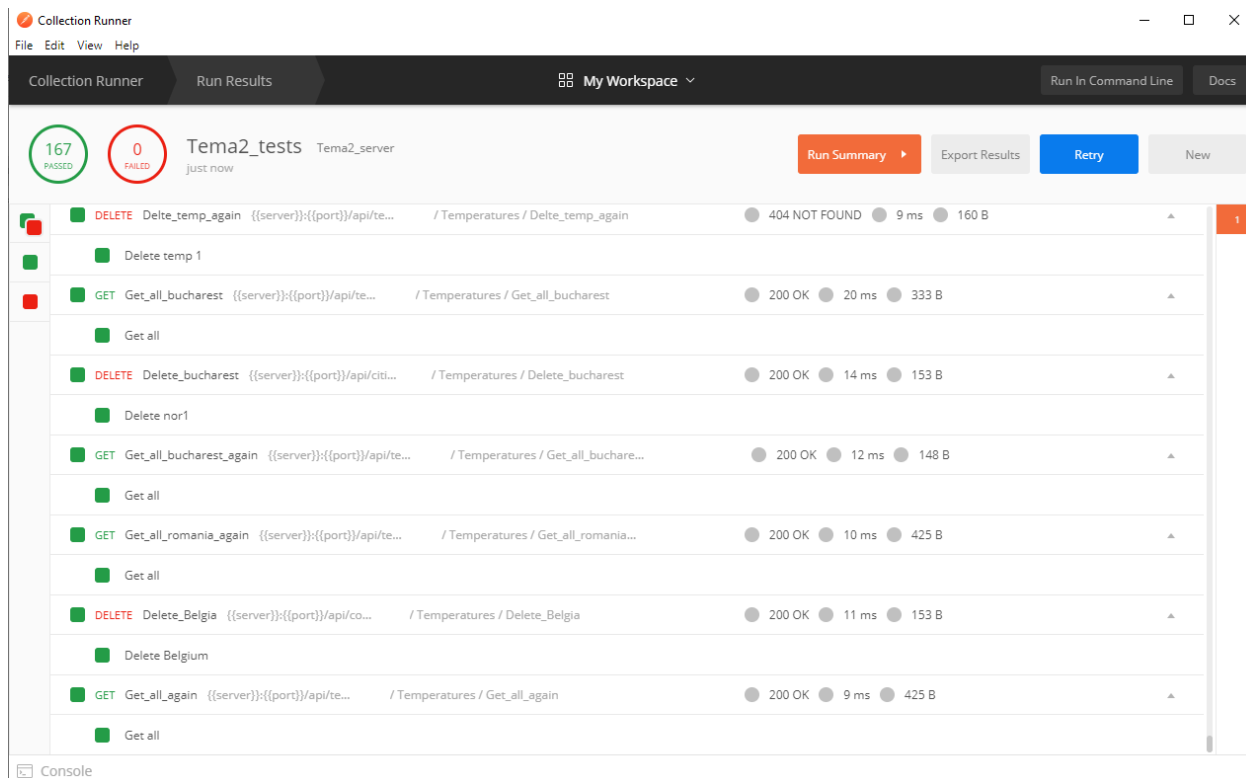
Cam asta e și partea de server. De unde știu că merge?

### 3 Testare

În arhiva temei am pus și o suită de 167 teste Postman pentru verificarea implementării temei. M-am chinuit să le fac cât mai cuprinzătoare. Testele trebuie rulate cu baza de date goală. Pentru asta, când le-am rulat am dat docker-compose .. down si docker volume prune pentru a mă asigura că totul pornește de la 0.

**!!Menționez că suita de teste am dat-o și altor persoane pentru a-și verifica tema!!**

În imaginea următoare se poate vedea rezultatul testelor. După cum se poate observa, toate trec cu brio.



Acum să le detaliez puțin. Testele sunt grupate în 3 foldere, specifice fiecărei tabele, anume countries, cities și temperatures. Fiecare folder conține teste pentru cererile grupate cum sunt și în enunț, în mare în ordinea în care sunt în enunț.

### 3.1 Countries

Primul test e un get care ar trebui să întoarcă lista vidă, baza de date fiind vidă. Urmează 3 teste post în care adaug în baza de date elemente, 3 țări. Urmează 7 teste de post în care fac toate combinațiile posibile de parametrii valizi, dar în care lipsesc unul, doi, sau toți 3. Toate ar trebui să pice cu 400. Urmează un test în care încerc să introduc o țară care are același nume cu una deja existentă și ar trebui să primească 409. Urmează 3 teste în care dau parametrii neconformi, pe rând. Toate ar trebui să pice cu 400. Urmează un get pentru a vedea ce e în baza de date, care ar trebui să întoarcă 3 lucruri(verificate în test).

Urmează testele de put. Prima dată fac un put valid, urmat de un put cu body valid, dar în care id-urile nu se potrivesc(body și cerere). Cel din urmă trebuie să crape cu 400. Urmează o cerere de update cu id invalid care trebuie să dea 404. După aceea vin 15 teste în care lipsesc unii parametrii și 4 în care tipul este greșit. Toate pică cu 400.

După put mai fac un get care arată datele modificate. După aceea șterg o țară(merge) și încerc imediat să șterg aceeași țară din nou(nu mai merge, 404) și fac un get ca să verific că țara a fost eliminată. La final adaug înapoi țara, mai adaug una pe deasupra și încerc un update cu nume duplicat la una dintre țări(desigur, trebuie să dea 409). La final verific să fie toate țările cum trebuie.

Astfel, toate cazurile posibile, eroare sau nu, din punctul meu de vedere, sunt acoperite.

### 3.2 Cities

Lucrurile merg întocmai ca la Countries. Fac get care întoarce nimic, urmat de 6 post-uri valide, 15 posturi nevalide pentru că lipsesc parametrii(400), un post nevalid pentru că se încearcă introducerea unui oraș deja existent( pereche id\_tara, nume\_oras existentă, 409), 4 posturi nevalide pentru că tipul este greșit(400) și în final un post nevalid pentru că țara de legătură nu există(404).

Urmează 3 cereri de get, una pe toate orașele, una pe toate orașele dintr-o țară și una cu id-ul unei țări care nu există.

Urmează o grămadă de teste put, prima validă urmată de o grămadă nevalide. Prima nevalidă are discrepanță de id-uri(400), a doua violează condiția de unicitate(409), a treia încearcă să updateze ceva ce nu există(404), urmate de 31 de teste în care lipsesc parametrii(400) și 5 în care parametrii au tip neconform(400).

Urmează două get-uri, unul pentru a vedea toate rezultatele și unul pentru a vedea rezultatele pentru Norvegia. Șterg, apoi un oraș din Norvegia(succes), încerc să-l șterg din nou(404) și verific rezultatele. După aceea șterg toată țara Norvegia(200), încerc s-o șterg din nou(404) și fac get pe toate orașele și pe orașele din Norvegia, cu asumția că au dispărut(**ștergere cascadată**). Ca să mă asigur mai dau un delete pe un oraș din norvegia(404).

### 3.3 Temperatures

Aici testarea automată e puțin mai greoaie din cauza timestampurilor. Am făcut automat tot ce se putea(tot în afară de get), pentru ca pe restul să le detaliez de mână în continuare. Fac 3 get-uri, unu per






total, unul cu orase si unul cu tari și mă asigur că nu întorc nimic. Inserez 7 temperaturi(200), inserez o temperatură care are corespunzător un oraș care nu există(404) și incerc cele 3 combinații posibile de parametrii lipsă(400) și 2 parametri cu tip nevalid(400). Nu pot testa 409 pentru că e imposibil să ai 409. Timestampul fiind inclusiv cu milisecunde **e imposibil să dai două cereri cu fix același timestamp** cât să crape unicitatea perechii (id\_oras, timestamp). Nici în put nu poți actualiza timestampul, deci nu văd cum ai putea face asta.

Urmează un put valid, urmat de un put nevalid cu 404, un put nevalid cu id-urile disparate, 7 put-uri cu parametrii lipsă în json(400) și 3 cu tipuri proaste(400). La final afișez din nou toate temperaturile, toate din București și toate din România. Șterg o temperatură(200) și încerc s-o șterg iarăși(404) și verific faptul că s-a șters. După aceea șterg orașul București și verific să fie un număr corect de intrări pentru temperaturile din București și din România(**ștergere cascadata**). După aceea șterg o țară(Belgia), și verific ștergerea cascadata și în acest caz.

Acestea au fost testele automate, din punctul meu cuprinzătoare pentru ce s-a scris în enunț și pentru ce am reușit să citesc pe forum. **Dacă lipsește vreun caz de eroare, chiar îmi pare rău, dar de aia am cerut teste oficiale....**

Acum, revenind la get-uri. După rularea testelor automate, baza de date ar trebui să arate cam așa:

Data Output	Explain	Messages	Notifications
id [PK] integer	nume_tara character varying (50)	latitudine double precision	longitudine double precision
1	3 Grecia	25.1	25.2
2	5 Romania	45.23	29.28

Data Output	Explain	Messages	Notifications	
 id [PK] integer	 id_tara integer	 nume_oras character varying (50)	 latitudine double precision	 longitudine double precision
1	3	3 Grec_city	20.85	20.1
2	5	5 Cluj	55.26	22.4

Data Output	Explain	Messages	Notifications
id [PK] integer	valoare double precision	timestamp timestamp without time zone	id_oras integer
1	3	26.6 2020-12-07 07:11:11.839817	5
2	5	23.1 2020-12-07 07:11:12.026378	5
3	6	24.5 2020-12-07 07:11:12.132084	5

Adaug inapoi Bucureștiul și un oraș în Grecia și pun câte o temperatură în fiecare.

Data Output	Explain	Messages	Notifications
id [PK] integer	valoare double precision	timestamp timestamp without time zone	id_oras integer
1	3	26.6 2020-12-07 07:11:11.839817	5
2	5	23.1 2020-12-07 07:11:12.026378	5
3	6	24.5 2020-12-07 07:11:12.132084	5
4	8	30.2 2020-12-07 07:56:13.915783	8
5	9	22.2 2020-12-07 07:56:52.752244	9

	id [PK] integer	id_tara integer	nume_oras character varying (50)	latitudine double precision	longitudine double precision	
1		3	3	Grec_city	20.85	20.15
2		5	5	Cluj	55.26	22.45
3		8	3	Atena	17.45	21.45
4		9	5	București	44.49	23.35

Acum e momentul de luat toate get-urile la rînd:

```
GET {{server}}:{{port}}/api/temperatures?lat=55.26&lon=22.45&from=2020-12-07&until=2020-12-30

1 [
2   {
3     "id": 3,
4     "timestamp": "2020-12-07 07:11:11.839817",
5     "valoare": 26.6
6   },
7   {
8     "id": 5,
9     "timestamp": "2020-12-07 07:11:12.026378",
10    "valoare": 23.1
11  },
12  {
13    "id": 6,
14    "timestamp": "2020-12-07 07:11:12.132084",
15    "valoare": 24.5
16  }
17 ]
```

Întoarce corect valorile corespunzătoare clujului. Dacă lipsesc parametrii, rezultatul este bun:

```
GET {{server}}:{{port}}/api/temperatures?lat=55.26&from=2020-12-07&until=2020-12-30

2 {
3   "id": 3,
4   "timestamp": "2020-12-07 07:11:11.839817",
5   "valoare": 26.6
6 },
7 {
8   "id": 5,
9   "timestamp": "2020-12-07 07:11:12.026378",
10  "valoare": 23.1
11 },
12 {
13   "id": 6,
14   "timestamp": "2020-12-07 07:11:12.132084",
15   "valoare": 24.5
16 }
```

```
GET {{server}}:{{port}}/api/temperatures?lon=22.45&from=2020-12-07&until=2020-12-30

3   "id": 3,
4   "timestamp": "2020-12-07 07:11:11.839817",
5   "valoare": 26.6
6 },
7 {
8   "id": 5,
9   "timestamp": "2020-12-07 07:11:12.026378",
10  "valoare": 23.1
11 },
12 {
13   "id": 6,
14   "timestamp": "2020-12-07 07:11:12.132084",
15   "valoare": 24.5
16 }
```



```
GET {{server}}:{{port}}/api/temperatures?lat=55.26&lon=22.45&until=2020-12-30
3      "id": 3,
4      "timestamp": "2020-12-07 07:11:11.839817",
5      "valoare": 26.6
6    },
7    {
8      "id": 5,
9      "timestamp": "2020-12-07 07:11:12.026378",
10     "valoare": 23.1
11   },
12   {
13     "id": 6,
14     "timestamp": "2020-12-07 07:11:12.132084",
15     "valoare": 24.5
16   }
17 }
```

```
GET {{server}}:{{port}}/api/temperatures?lat=55.26&lon=22.45&from=2020-12-07
3      "id": 3,
4      "timestamp": "2020-12-07 07:11:11.839817",
5      "valoare": 26.6
6    },
7    {
8      "id": 5,
9      "timestamp": "2020-12-07 07:11:12.026378",
10     "valoare": 23.1
11   },
12   {
13     "id": 6,
14     "timestamp": "2020-12-07 07:11:12.132084",
15     "valoare": 24.5
16   }
17 }
```

```
GET {{server}}:{{port}}/api/temperatures?lat=55.26&lon=22.45
1    {
2      {
3        "id": 3,
4        "timestamp": "2020-12-07 07:11:11.839817",
5        "valoare": 26.6
6      },
7      {
8        "id": 5,
9        "timestamp": "2020-12-07 07:11:12.026378",
10       "valoare": 23.1
11      },
12      {
13        "id": 6,
14        "timestamp": "2020-12-07 07:11:12.132084",
15        "valoare": 24.5
16      }
17    }
18  }
```

```
GET {{server}}:{{port}}/api/temperatures?from=2020-12-07&until=2020-12-30
```

```
1    {
2      {
3        "id": 3,
4        "timestamp": "2020-12-07 07:11:11.839817",
5        "valoare": 26.6
6      },
7      {
8        "id": 5,
9        "timestamp": "2020-12-07 07:11:12.026378",
10       "valoare": 23.1
11      },
12      {
13        "id": 6,
14        "timestamp": "2020-12-07 07:11:12.132084",
15        "valoare": 24.5
16      },
17      {
18        "id": 8,
19        "timestamp": "2020-12-07 07:56:13.915783",
20        "valoare": 30.2
21      },
22      {
23        "id": 9,
24        "timestamp": "2020-12-07 07:56:52.752244",
25        "valoare": 22.2
26      }
27    }
28  }
```

```
GET {{server}}:{{port}}/api/temperatures
```

```
      "id": 3,
      "timestamp": "2020-12-07 07:11:11.839817",
      "valoare": 26.6
    },
    {
      "id": 5,
      "timestamp": "2020-12-07 07:11:12.026378",
      "valoare": 23.1
    },
    {
      "id": 6,
      "timestamp": "2020-12-07 07:11:12.132084",
      "valoare": 24.5
    },
    {
      "id": 8,
      "timestamp": "2020-12-07 07:56:13.915783",
      "valoare": 30.2
    },
    {
      "id": 9,
      "timestamp": "2020-12-07 07:56:52.752244",
      "valoare": 22.2
    }
  }
```

Dacă dăm parametrii nevalizi, sau nu sunt datele în ordinea bună, răspunsul va fi 200 și lista vidă:

GET

▼

{{server}}:{{port}}/api/temperatures?lat=hehe&lon=22.45&from=2020-12-07&until=2020-12-30

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

Visualize

JSON

1

[ ]

22.45

2020-12-07

GET

▼

{{server}}:{{port}}/api/temperatures?lat=hehe&lon=hehe&from=2020-12-07&until=2020-12-30

Pretty

Raw

Preview

Visualize

JS

Body

Pre-request Script

Tests

Settings

1

[ ]

hehe

GET

▼

{{server}}:{{port}}/api/temperatures?lat=55.26&lon=22.45&from=2020-12-30&until=2020-12-07

Pretty

Raw

Preview

Visualize

Body

Pre-request Script

Tests

Settings

1

[ ]

22.45

Aceleași lucruri pentru get id oras:

GET	▼	{{server}}:{{port}}/api/temperatures/cities/8?from=2020-12-07&until=2020-12-30
3		"id": 8,
4		"timestamp": "2020-12-07 08:17:06.828469",
5		"valoare": 30.2
6		}
7		[ ]

GET	▼	{{server}}:{{port}}/api/temperatures/cities/9?from=2020-12-07&until=2020-12-30
3		"id": 9,
4		"timestamp": "2020-12-07 08:17:09.293622",
5		"valoare": 22.2
6		}
7		[ ]

```
GET {{server}}:{{port}}/api/temperatures/cities/5?from=2020-12-07&until=2020-12-30
{
  "id": 3,
  "timestamp": "2020-12-07 08:16:41.441634",
  "valoare": 26.6
},
{
  "id": 5,
  "timestamp": "2020-12-07 08:16:41.631792",
  "valoare": 23.1
},
{
  "id": 6,
  "timestamp": "2020-12-07 08:16:41.727126",
  "valoare": 24.5
}
```

```
GET {{server}}:{{port}}/api/temperatures/cities/5?from=2020-12-07 08:16:41.6&until=2020-12-30
{
  "id": 5,
  "timestamp": "2020-12-07 08:16:41.631792",
  "valoare": 23.1
},
{
  "id": 6,
  "timestamp": "2020-12-07 08:16:41.727126",
  "valoare": 24.5
}
```

```
GET {{server}}:{{port}}/api/temperatures/cities/5?from=2020-12-07 08:16&until=2020-12-07 08:16:41.5
{
  "id": 3,
  "timestamp": "2020-12-07 08:16:41.441634",
  "valoare": 26.6
}
```

```
GET {{server}}:{{port}}/api/temperatures/cities/5?until=2020-12-07 08:16:41.7
{
  "id": 3,
  "timestamp": "2020-12-07 08:16:41.441634",
  "valoare": 26.6
},
{
  "id": 5,
  "timestamp": "2020-12-07 08:16:41.631792",
  "valoare": 23.1
}
```

```
GET {{server}}:{{port}}/api/temperatures/cities/5?from=2020-12-07 08:16:41.6

3      "id": 5,
4      "timestamp": "2020-12-07 08:16:41.631792",
5      "valoare": 23.1
6    },
7    {
8      "id": 6,
9      "timestamp": "2020-12-07 08:16:41.727126",
10     "valoare": 24.5
11   }
12 ]
```

```
GET {{server}}:{{port}}/api/temperatures/cities/5?from=2020-12-30&until=2020-12-07
```

1 

```
GET {{server}}:{{port}}/api/temperatures/cities/5

      "id": 3,
      "timestamp": "2020-12-07 08:16:41.441634",
      "valoare": 26.6
    },
    {
      "id": 5,
      "timestamp": "2020-12-07 08:16:41.631792",
      "valoare": 23.1
    },
    {
      "id": 6,
      "timestamp": "2020-12-07 08:16:41.727126",
      "valoare": 24.5
    }
  ]
```

Si id tara:

```
GET {{server}}:{{port}}/api/temperatures/countries/5

3      "id": 3,
4      "timestamp": "2020-12-07 08:16:41.441634",
5      "valoare": 26.6
6    },
7    {
8      "id": 5,
9      "timestamp": "2020-12-07 08:16:41.631792",
10     "valoare": 23.1
11   },
12   {
13     "id": 6,
14     "timestamp": "2020-12-07 08:16:41.727126",
15     "valoare": 24.5
16   },
17   {
18     "id": 9,
19     "timestamp": "2020-12-07 08:17:09.293622",
20     "valoare": 22.2
21   }
22 ]
```

```
GET {{server}}:{{port}}/api/temperatures/countries/5?from=2020-12-07 08:16&until=2020-12-07 08:16:41.7

2 {
3   "id": 3,
4   "timestamp": "2020-12-07 08:16:41.441634",
5   "valoare": 26.6
6 },
7 {
8   "id": 5,
9   "timestamp": "2020-12-07 08:16:41.631792",
10  "valoare": 23.1
11 }
12 ]
```

```
GET {{server}}:{{port}}/api/temperatures/countries/5?from=2021-12-07 08:16&until=2020-12-07 08:16:41.7 1 [ ]
```

```
GET {{server}}:{{port}}/api/temperatures/countries/5?from=2020-12-07 08:16:42

3 {
4   "id": 9,
5   "timestamp": "2020-12-07 08:17:09.293622",
6   "valoare": 22.2
7 }
```

```
GET {{server}}:{{port}}/api/temperatures/countries/5?until=2020-12-07 08:16:41.7

3 {
4   "id": 3,
5   "timestamp": "2020-12-07 08:16:41.441634",
6   "valoare": 26.6
7 },
8 {
9   "id": 5,
10  "timestamp": "2020-12-07 08:16:41.631792",
11  "valoare": 23.1
12 }
```

## 4 Concluzii

Și cam asta e toată tema, implementată și testată în detaliu. Sper că totul va fi în regulă și că nu vor fi probleme la corectare. Cred că am documentat cum trebuie și că am acoperit toate cerințele temei.