Trabalho Prático - REST API com Python e Flask

Ricardo Daniel da Silva Teixeira - Nº 20080

Bruna Raquel Noversa Macieira – Nº 21139

Francisco Rafael Dias Pereira – Nº 21156

Comunicações de Dados

Professor Miguel Lopes

Ano letivo 2021/2022

Licenciatura em Engenharia de Sistemas Informáticos

Escola Superior de Tecnologia

Instituto Politécnico do Cávado e do Ave

RESUMO

O presente relatório foi elaborado no âmbito da Unidade Curricular de Comunicações de Dados da Licenciatura de Engenharia de Sistemas Informáticos.

Este documento regista e fundamenta as decisões tomadas ao longo do desenvolvimento do projeto em questão, assim como demonstrações do projeto em questão.

O trabalho prático consiste, de forma sucinta, na realização de uma REST API, desenvolvida em *Python* em conjunto com a *framework Flask*, utilizada pela aplicação para realizar os pontos a seguir descritos. Apresentará, também, uma interface ao utilizador, desenvolvida em HTML. Todos estes serviços estarão disponíveis através da *web.*

A base de dados foi criada em *MongoDB* e foi ainda utilizado *JTW* (*JSON Web Token*), para que as trocas de dados fossem realizadas de forma segura.

ÍNDICE

Conteúdo

1. Introdução ao problema abordado

Descrição do problema

O problema descrito no enunciado é o *job shop*, que se baseia na capacidade de várias máquinas realizarem operações.

Para se produzir um trabalho (*job*), é necessário realizar um conjunto de operações, por sequência, em várias máquinas. Como as máquinas não conseguem realizar todas as operações, cada trabalho é uma sequência de operações, em que cada operação tem de ser feita numa máquina específica.

Adicionalmente, porque cada trabalho é distinto dos restantes, a sequência de máquinas de cada trabalho pode não ser a mesma dos restantes trabalhos (cada trabalho tem a sua própria sequência de máquinas).

Contudo, a ordem das máquinas para cada trabalho tem de ser respeitada (uma operação de um trabalho que deve ser executada numa máquina específica só pode começar quando a operação anterior desse mesmo trabalho já terminou, assim como a operação seguinte do trabalho só pode começar depois da operação atual terminar).

Realizar uma operação numa máquina demora tempo, que pode ser distinto para cada operação. Uma máquina só consegue fazer uma operação de cada vez. Assim, quando começa uma operação, só fica livre para novas operações após esta terminar.

```python
from flask import Flask
from bson.json_util import ObjectId
import json
from users import users
from simulation import simulation
from jobshop import jobshop


class MyEncoder(json.JSONEncoder):

    def default(self, obj):
        if isinstance(obj, ObjectId):
            return str(obj)
        return super(MyEncoder, self).default(obj)


app = Flask(__name__)
app.json_encoder = MyEncoder
app.register_blueprint(users)
app.register_blueprint(simulation)
app.register_blueprint(jobshop)
```

Figura 1 - app.py

```python
from pymongo import MongoClient
import certifi

def get_database():

    # Provide the mongodb atlas url to connect python to mongodb using pymongo
    CONNECTION_STRING = "mongodb+srv://admin:admin@cluster0.x9mgy.mongodb.net/myFirstDatabase?retryWrites=true&w=majority"

    # Create a connection using MongoClient. You can import MongoClient or use pymongo.MongoClient
    client = MongoClient(CONNECTION_STRING, tlsCAFile=certifi.where())

    # Create the database for our example (we will use the same database throughout the tutorial
    return client.JobsDB
```

Figura 2 - db.py

Objetivos do projeto

O objetivo da gestão do plano de produção é produzir uma lista de operações, cada uma para ser realizada numa máquina num determinado instante. Cada alínea tem um

objetivo, sendo cada um deles abordado em seguida e demonstrada a sua resolução, equacionada de acordo com a interpretação do grupo e maior eficácia do projeto.

2. Serviço de Gestão de Simulações

Para esta alínea, foi desenvolvida uma API, utilizando a linguagem de programação *Python* e a *framework Flask*, capaz de gerir simulações, ou seja, um plano de produção capaz de controlar máquinas, de modo a estas produzirem uma lista de produtos.

```
simulation = Blueprint('simulation', __name__)
```

Figura 3 - simulação usando a library Blueprint

Tal deveria ser capaz de criar uma simulação, com os parâmetros necessários (número de máquinas, número de trabalhos e número de operações), listá-las e removê-las, caso o utilizador o pretendesse.

```
# Add simulation
# {
#     "author_id": "naewfiunfn3f023jf092"
#     "nr_operations": 3,
#     "nr_jobs": 3,
#     "nr_machines": 3,
#     "is_active": true,
#     "table": []
#     "production": []
# }
@simulation.post("/simulation")
@token_required
def add_simulation(f):
    if request.is_json:
        simulation_request = request.get_json()

        db.Simulation.insert_one(simulation_request)
        return jsonify(simulation_request), 201
    return {"error": "Request must be JSON"}, 415
```

Figura 4 - adicionar simulação

Os erros HTTP 201 e 415 significam, respetivamente, “Created” e “Unsupported Media Type”, que são retornados quando a simulação é criada ou, caso contrário, quando o *request* não é *JSON*.

Remover uma simulação não implica a sua anulação, apenas não seria possível editá-la, podendo, assim, ser visualizada. Esta ação decorre do facto de ser boa prática não eliminar permanentemente dados, pois pode perder-se informação importante.

```python
@simulation.delete("/simulation")
@token_required
def remove_simulation(f):
    if request.is_json:
        data = request.get_json()
        filter_id = {'_id': ObjectId(data["_id"])}
        new_values = {"$set": {"isActive": bool(False)}}
        update_result = db.Simulation.update_one(filter_id, new_values)
        if update_result.modified_count == 1:
            return {"msg": "Simulation removed successfully!"}, 200
        else:
            return {"error": "Not found"}, 404
    return {"error": "Request must be JSON"}, 415
```

Figura 5 - eliminar simulação

O erro HTTP 404 significa “Not Found”.

3. Serviço de Tabela de Produção

Para esta alínea, foi desenvolvida uma API, utilizando a linguagem de programação *Python* e a *framework Flask*, capaz de gerir tabelas de produção simulações. Cada simulação é definida, como vimos acima, por um conjunto de parâmetros, sendo esses máquinas, trabalhos e respetivas operações e uma tabela de produção que demonstra, para cada operação dum determinado trabalho, qual a máquina que a consegue realizar e o tempo respetivo de duração. Como exemplo, foi utilizado o serviço prestado pela aplicação *OR-Tools* da *Google*.

Tal deveria ser capaz de: criar uma tabela com os parâmetros descritos; indicar o término da construção da tabela, indicando se a operação foi bem sucedida ou não, devendo ainda ser possível alterar os valores de cada operação, mas não introduzir

novos valores; consultar a tabela, através de uma consulta individual para cada operação de cada trabalho, retornando os restantes parâmetros e indicar erro caso a operação não esteja introduzida; fazer download de um ficheiro de texto representativo da tabela.

```python
# {
#     "_id": "8j298fifjweoifj",
#     "table": [
#         [[1, 3], [0, 2]]
#         [[0, 2], [1, 3]]
#     ]
# }
@simulation.put("/simulation")
@token_required
def update_data_simulation(f):
    if request.is_json:
        data = request.get_json()
        filter_id = {'_id': ObjectId(data["_id"])}
        table = data["table"]
        values = db.Simulation.find_one(filter_id)
        verify = False
        counter = 0
        for job in data["table"]:
            if job is not False:
                counter += 1
                if len(job) == values["nr_operations"]:
                    verify = True

        if values["nr_jobs"] != counter:
            return {"error": "All jobs should have at least one operation!"}, 400

        if not verify:
            return {"error": "At least one operation is empty!"}, 400

        production = jobshop.jobshop_resolver(table)
        update = {"$set": {"table": table, "production": production}}
        update_result = db.Simulation.update_one(filter_id, update)
        if update_result.modified_count == 1:
            return {"msg": "Table updated successfully!"}, 200

    return {"error": "Request must be JSON"}, 415
```

Figura 6 - atualiza dados da simulação manualmente

4. Serviço de Plano de Produção

Para esta alínea, foi desenvolvida uma API, utilizando a linguagem de programação *Python* e a *framework Flask*, capaz de definir um plano de produção, baseado numa lista que atribui um tempo de início a cada operação de cada trabalho.

No entanto, as operações de cada trabalho têm de ser realizadas por ordem sequencial, apenas começando a operação seguinte quando a respetiva já estiver concluída. Além disso, cada máquina apenas pode executar uma operação de cada vez.

Tal deveria ser capaz de: atribuir um tempo de início de uma operação indicado manualmente pelo utilizador, enviando uma resposta de erro caso o tempo de início de uma operação colida com o tempo de resolução da operação anterior do mesmo trabalho ou se o tempo de início da operação escolhida começar numa máquina que esteja ocupada no tempo proposto; validar se todas as operações têm um tempo de início associado; indicar o tempo de conclusão da operação apresentada em último lugar; fazer download de um ficheiro textual com o plano de produção.

```
{
    "_id": "oenfowiehf43984f"   -> simulation ID to insert plan table
    "production": [
            {
            "duration_time": 2,
            "end_time": 2,
            "job_id": 1,
            "machine": 0,
            "start_time": 0,
            "task_id": 0
            }
     ]
}
@simulation.put("/simulationManualUpdate")
def update_manual_simmulation(f):
    if request.is_json:
        simulation_insert = request.get_json()
        simulation_element = db.Simulation.find_one({"_id": ObjectId(simulation_insert["_id"])})
        if simulation_element is not None:
            conflict = False
            for production in simulation_insert["production"]:
                for operation in simulation_insert["production"]:
                    if production["job_id"] == operation["job_id"] and production["task_id"] == operation["task_id"] and production["durati
                        continue
                    if operation["job_id"] == production["job_id"] and operation["end_time"] > production["start_time"]:
                        conflict = True
                        break

                    i1 = pd.Interval(operation["start_time"], operation["end_time"])
                    i2 = pd.Interval(production["start_time"], production["end_time"])

                    if production["machine"] == operation["machine"] and i1.overlaps(i2):
                        conflict = True
                        break

                    if conflict:
                        return {"Error": "Operation start and end time, overlaps other operation"}, 409
            table = simulation_element["production"]
            update = {"$set": {"production": table}}
            filter_id = {'_id': ObjectId(simulation_element["_id"])}
            update_result = db.Simulation.update_one(filter_id, update)
        return jsonify(update_result), 200
    return {"error": "Request must be JSON"}, 415
```

Figura 7 - atualiza dados de produção manualmente

5. Serviço de Plano de Produção Automatizado

Para esta alínea, foi desenvolvida uma API, utilizando a linguagem de programação *Python* e a *framework Flask*, capaz de definir um plano de produção automatizado. Foi utilizado, como base, um exemplo da biblioteca *OR-Tools* da *Google*, para gerir automaticamente uma solução de plano de produção.

Tal deveria ser capaz de: atribuir um tempo de início de uma operação indicado automaticamente pela ferramenta; indicar o tempo de conclusão da operação apresentada em último lugar; fazer download de um ficheiro textual com o plano de produção.

```python
jobshop = Blueprint('jobshop', __name__)


def jobshop_resolver(jobs_data):
    """Minimal jobshop problem."""
    # Data.
    # jobs_data = [  # task = (machine_id, processing_time).
    #     [(0, 3), (1, 2), (2, 2)],  # Job0
    #     [(0, 2), (2, 1), (1, 4)],  # Job1
    #     [(1, 4), (2, 3)]  # Job2
    # ]

    # Data structure to return
    automated_simulation = list()

    machines_count = 1 + max(task[0] for job in jobs_data for task in job)
    all_machines = range(machines_count)
    # Computes horizon dynamically as the sum of all durations.
    horizon = sum(task[1] for job in jobs_data for task in job)

    # Create the model.
    model = cp_model.CpModel()

    # Named tuple to store information about created variables.
    task_type = collections.namedtuple('task_type', 'start end interval')
    # Named tuple to manipulate solution information.
    assigned_task_type = collections.namedtuple('assigned_task_type',
                                                'start job index duration')
```

Figura 8 - jobshop.py (1)

```
# Creates job intervals and add to the corresponding machine lists.
all_tasks = {}
machine_to_intervals = collections.defaultdict(list)

for job_id, job in enumerate(jobs_data):
    for task_id, task in enumerate(job):
        machine = task[0]
        duration = task[1]
        suffix = '_%i_%i' % (job_id, task_id)
        start_var = model.NewIntVar(0, horizon, 'start' + suffix)
        end_var = model.NewIntVar(0, horizon, 'end' + suffix)
        interval_var = model.NewIntervalVar(start_var, duration, end_var,
                                            'interval' + suffix)
        all_tasks[job_id, task_id] = task_type(start=start_var,
                                               end=end_var,
                                               interval=interval_var)
        machine_to_intervals[machine].append(interval_var)

# Create and add disjunctive constraints.
for machine in all_machines:
    model.AddNoOverlap(machine_to_intervals[machine])

# Precedences inside a job.
for job_id, job in enumerate(jobs_data):
    for task_id in range(len(job) - 1):
        model.Add(all_tasks[job_id, task_id +
                            1].start >= all_tasks[job_id, task_id].end)

# Makespan objective.
obj_var = model.NewIntVar(0, horizon, 'makespan')
model.AddMaxEquality(obj_var, [
    all_tasks[job_id, len(job) - 1].end
    for job_id, job in enumerate(jobs_data)
])
model.Minimize(obj_var)

# Creates the solver and solve.
solver = cp_model.CpSolver()
status = solver.Solve(model)
```

Figura 9 - jobshop.py (2)

```
if status == cp_model.OPTIMAL or status == cp_model.FEASIBLE:
    print('Solution:')
    # Create one list of assigned tasks per machine.
    assigned_jobs = collections.defaultdict(list)
    for job_id, job in enumerate(jobs_data):
        for task_id, task in enumerate(job):
            machine = task[0]
            assigned_jobs[machine].append(
                assigned_task_type(start=solver.Value(
                    all_tasks[job_id, task_id].start),
                                   job=job_id,
                                   index=task_id,
                                   duration=task[1]))

    # Create per machine output lines.
    output = ''
    for machine in all_machines:
        # Sort by starting time.
        assigned_jobs[machine].sort()
        sol_line_tasks = 'Machine ' + str(machine) + ': '
        sol_line = '           '

        for assigned_task in assigned_jobs[machine]:
            name = 'job_%i_task_%i' % (assigned_task.job,
                                       assigned_task.index)

            # Add operation (job_id + task_id) to data structure
            automated_simulation.append(
                {
                    "job_id": assigned_task.job,
                    "task_id": assigned_task.index,
                    "machine": machine,
                    "start_time": assigned_task.start,
                    "end_time": assigned_task.start + assigned_task.duration,
                    "duration_time": assigned_task.duration
                }
            )

            # Add spaces to output to align columns.
            sol_line_tasks += '%-15s' % name
```

Figura 10 - jobshop.py (3)

```
                start = assigned_task.start
                duration = assigned_task.duration
                sol_tmp = '[%i,%i]' % (start, start + duration)
                # Add spaces to output to align columns.
                sol_line += '%-15s' % sol_tmp

            sol_line += '\n'
            sol_line_tasks += '\n'
            output += sol_line_tasks
            output += sol_line

        # Finally print the solution found.
        print(f'Optimal Schedule Length: {solver.ObjectiveValue()}')
        print(output)
        data = {"optimal_time": solver.ObjectiveValue(),
                "output": output,
                "conflicts": solver.NumConflicts(),
                "branches": solver.NumBranches(),
                "wall_time": solver.WallTime()}
    else:
        print('No solution found.')

    # Statistics.
    print('\nStatistics')
    print('  - conflicts: %i' % solver.NumConflicts())
    print('  - branches : %i' % solver.NumBranches())
    print('  - wall time: %f s' % solver.WallTime())

    return automated_simulation

if __name__ == '__main__':
    jobshop()
```

Figura 11 - jobshop.py (4)

6. Serviço de Gestão de Utilizadores

Para esta alínea, foi desenvolvida uma API, utilizando a linguagem de programação *Python* e a *framework Flask*, capaz de definir um plano de gestão de utilizadores.

O administrador, responsável por essa gestão, deveria ser capaz de: criar utilizadores, atribuindo uma palavra-passe individual inicial; permitir alterar a palavra-passe de um utilizador, sendo que esse utilizador é que decide a palavra-passe pretendida, sendo, por fim, aprovada ou não pelo administrador; remover um utilizador. Remover um utilizador não implica a anulação dos seus dados, apenas não seria possível editá-los, podendo, assim, ser visualizados pelo administrador. Esta ação decorre do facto de ser

boa prática não eliminar permanentemente dados, pois pode perder-se informação importante e, ainda, ter consequências graves ao nível da gestão da base de dados.

Para a resolução deste problema, foi utilizado *JTW* (*JSON Web Token*). Este *token* de acesso torna possível a troca de dados segura entre duas entidades, não havendo trocas com as *queries* da base de dados e a sessão não fica necessariamente guardada no servidor (as aplicações *REST* são *stateless*, pois a informação necessária para a autenticação é enviada com o *request*). As mensagens trocadas são encriptadas, de modo a fornecer segurança, e os utilizadores não têm contacto com o *token* em si; apenas há interação entre o cliente e o servidor.

O *JWT* encontra-se dividido em *header*, *payload* e assinatura.

O *header* tem duas partes: o tipo do *token* e o algoritmo de encriptação utilizado. Neste caso, o algoritmo de encriptação utilizado é o *HS256*.

O *payload* guarda a informação que será transmitida, que está presente como chaves ou pares de valores.

A *secret key* é verificada na assinatura do *JSON*, de modo a controlar alterações na mensagem. Caso nada tenha sido alterado, os dados estão seguros.

```python
# Requires token to auth
def token_required(f):
    @wraps(f)
    def decorator(*args, **kwargs):
        token = None
        if 'x-access-tokens' in request.headers:
            token = request.headers['x-access-tokens']

        if not token:
            return jsonify({'message': 'a valid token is missing'})
        try:
            data = jwt.decode(token, "b893510d0644a24ea8e8030fa7ec13ec", algorithms=["HS256"])
            current_user = db.User.find_one({"_id": ObjectId(data["_id"])})
        except:
            return jsonify({'message': 'token is invalid'})

        return f(current_user, *args, **kwargs)
    return decorator
```

Figura 12 - autorização

```python
# Verify if is admin
def admin_required(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        if 'x-access-tokens' in request.headers:
            token = request.headers['x-access-tokens']

        if not token:
            return jsonify({'message': 'a valid token is missing'}), 401

        payload_data = jwt.decode(token, "b893510d0644a24ea8e8030fa7ec13ec", algorithms=["HS256"])

        if payload_data['admin'] is False:
            return jsonify(msg='Admins only!'), 401
        else:
            return fn(*args, **kwargs)
    return wrapper
```

Figura 13 - verifica se é o administrador da página

```python
# {
#     "_id": "jekjhf",
#     "current_password": "123",
#     "new_password": "872387"
# }
# Set new user password
@users.put("/password")
@token_required
def set_new_password_user():
    if request.is_json:
        data = request.get_json()
        token = None
        if 'x-access-tokens' in request.headers:
            token = request.headers['x-access-tokens']
        user_data = db.User.find_one({"_id": ObjectId(data["_id"])})
        if not token:
            return jsonify({'message': 'a valid token is missing'})
        try:
            payload_data = jwt.decode(token, secret, algorithms=["HS256"])
            if (payload_data["_id"] == ObjectId(data["_id"])) and check_password_hash(user_data["password"], data["current_password"]) and user_data["permission_pw"]:
                new_values = {"$set": {"password": generate_password_hash(data["new_password"], method='sha256')}}
                filter_id = {'_id': ObjectId(data["_id"])}
                nr_rows_affected = db.User.update_one(filter_id, new_values)
                if nr_rows_affected.count() == 1:
                    return {"msg": "Password changed successfully!"}, 200
                else:
                    return {"error": "User not found or not allowed to change the password!"}, 404
        except:
            return jsonify({'message': 'token is invalid'})
    return {"error": "Request must be JSON"}, 415
```

Figura 14 - atualizar password do utilizador

```python
# {
#     "_id": "kjkjhkjh"
# }
@users.delete("/user")
@token_required
@admin_required
def remove_user():
    if request.is_json:
        data = request.get_json()
        filter_id = {'_id': ObjectId(data["_id"])}
        new_values = {"$set": {"isActive": bool(False)}}
        update_result = db.User.update_one(filter_id, new_values)
        filter_id = {"author_id": filter_id}
        new_values = {"$set": {"isActive": bool(False)}}
        db.Simulation.update_many(filter_id, new_values)

        if update_result.modified_count == 1:
            return {"msg": "User deleted successfully!"}, 200
        else:
            return {"error": "Not found"}, 404
    return {"error": "Request must be JSON"}, 415
```

Figura 15 - eliminar utilizador

```python
import datetime
from functools import wraps
from time import timezone
import jwt
from bson.json_util import ObjectId
from flask import Blueprint
from flask import request, jsonify, make_response
from werkzeug.security import generate_password_hash, check_password_hash
import app
from db import get_database
import os
from json import dumps

secret = os.environ.get("secret-key")

users = Blueprint('users', __name__)

db = get_database()
```

Figura 16 - bibliotecas e outros do ficheiro users

7. Conclusão

Em suma, o projeto desenvolvido é visto como uma ferramenta bastante útil para o mercado de trabalho, sendo crucial para a boa gestão de tempo e recursos de uma empresa.

Apesar de ter ficado um pouco aquém das expectativas, o grupo faz uma avaliação positiva deste trabalho, tendo em conta a sua execução e valor.