

PokeMap

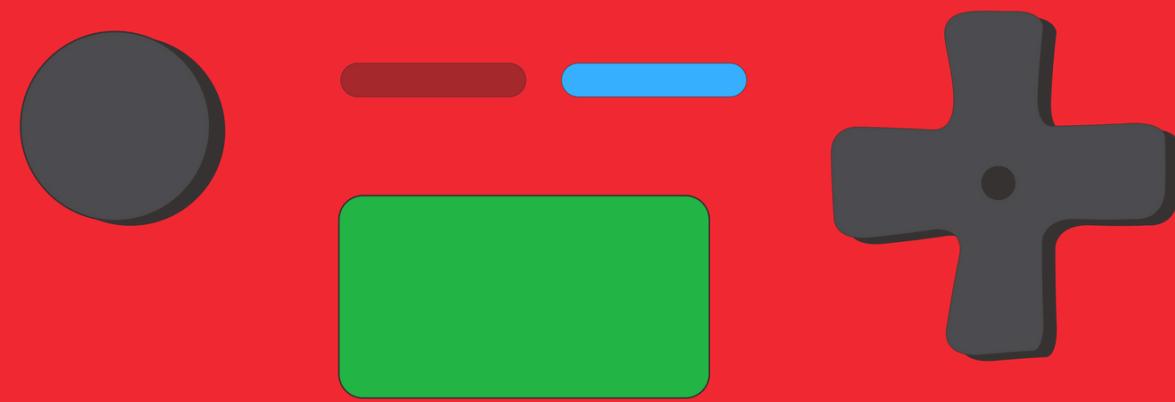


P2

Gabriel Barbosa
Gabriel Meira
Gabriela Fiori
Heitor Morais
Henrique Cesar



- #001 Consumo de APIs
- #002 Estratégias de Cache
- #003 Autenticação com JWT
- #004 Integração com o MongoDB
- #005 Aplicação



DEFINIÇÃO E IMPORTÂNCIA DE APIs



API, sigla para "Application Programming Interface" (Interface de Programação de Aplicações, em português), é um conjunto de definições e protocolos que permitem a comunicação entre diferentes sistemas de software



Endpoints

URLs que representam recursos específicos com os quais a aplicação pode interagir.

Métodos HTTP

Ações que podem ser executadas nos endpoints, como GET, POST, PUT, DELETE.

Parâmetros

Dados que podem ser enviados ou recebidos pela API para interagir com os recursos.

Respostas

Dados retornados pela API após a execução de uma ação.



Tipos de APIs

REST

As mais comuns, utilizam a web como base para comunicação, seguindo princípios como recursos, representações e métodos HTTP.

SOAP

Utilizam protocolos XML e web services para troca de dados estruturados.

RPC

Permitem a execução remota de procedimentos em outros sistemas.

SO

Fornecem acesso a recursos e serviços do sistema operacional, como gerenciamento de arquivos e memória..



Realizando chamadas a APIs com Python



Envolve a utilização de bibliotecas que permitem enviar requisições HTTP e manipular as respostas. Uma das bibliotecas mais populares para esse propósito é o requests.



Requisição GET

```
import requests

# URL da API
url = 'https://api.exemplo.com/dados'

# Realizando a requisição GET
response = requests.get(url)

# Verificando o status da resposta
if response.status_code == 200:
    # Convertendo a resposta em JSON
    data = response.json()
    print(data)
else:
    print(f"Erro na requisição: {response.status_code}")
```



Requisição POST

```
import requests

# URL da API
url = 'https://api.exemplo.com/envia-dados'

# Dados a serem enviados
payload = {
    'nome': 'João',
    'idade': 30
}

# Cabeçalhos (opcional)
headers = {
    'Content-Type': 'application/json'
}

# Realizando a requisição POST
response = requests.post(url, json=payload, headers=headers)

# Verificando o status da resposta
if response.status_code == 201:
    # Convertendo a resposta em JSON
    data = response.json()
    print(data)
else:
    print(f"Erro na requisição: {response.status_code}")
|
```



Parâmetros na URL

```
import requests

url = 'https://api.exemplo.com/busca'

# Parâmetros da URL
params = {
    'query': 'python',
    'page': 2
}

response = requests.get(url, params=params)

if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print(f"Erro na requisição: {response.status_code}")
```



Autenticação

```
import requests
from requests.auth import HTTPBasicAuth

url = 'https://api.exemplo.com/privado'

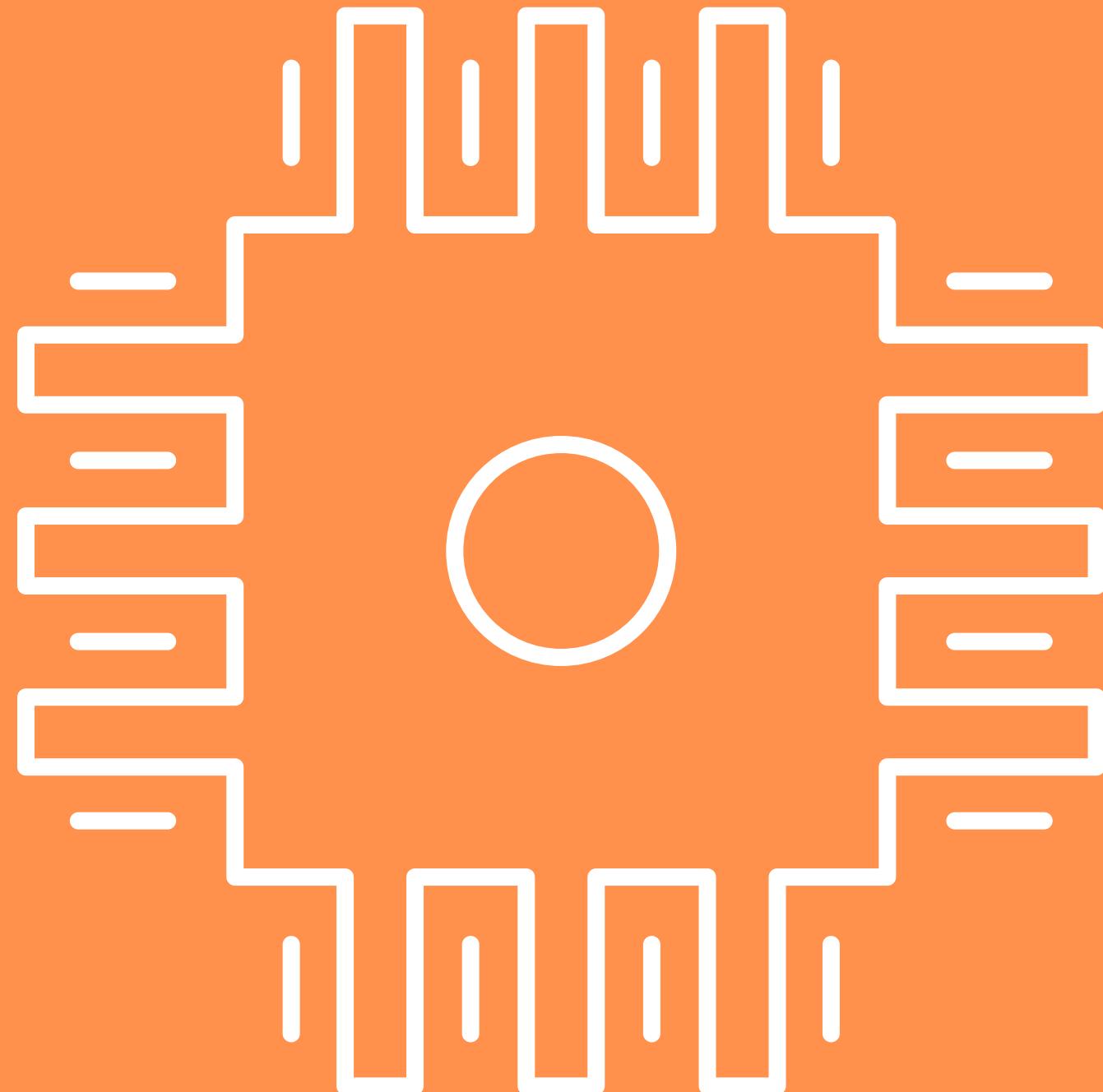
# Credenciais de autenticação
username = 'usuario'
password = 'senha'

response = requests.get(url, auth=HTTPBasicAuth(username, password))

if response.status_code == 200:
    data = response.json()
    print(data)
else:
    print(f"Erro na requisição: {response.status_code}")
```



Estratégias de cache



O cache é uma técnica de armazenamento temporário de dados que permite o acesso rápido e eficiente a informações frequentemente acessadas.



Estratégias de cache

1. Cache de API

Armazenar temporariamente os resultados de consultas ou solicitações feitas a uma API

Em vez de enviar uma nova solicitação à API sempre que os mesmos dados forem necessários, o cache armazena a resposta da solicitação anterior por um período de tempo determinado.

Pontos a se considerar

- Validade dos dados em cache
- Capacidade de lidar com dados atualizados
- Segurança das informações armazenadas



Estratégias de cache

2. Cache de Dados de Sessão

Armazena informações sobre a sessão do usuário, como autenticação e preferências

Evita consultas repetidas ao banco de dados durante a sessão do usuário. Isso melhora a experiência do usuário e reduz a carga no servidor de banco de dados.

Pontos a se considerar

- Problemas de consistência de dados se os dados armazenados em cache estiverem desatualizados
- Gerenciamento de Expiração
- Pode representar um risco de segurança se informações sensíveis, como tokens de autenticação ou dados pessoais



Estratégias de cache

3. Cache de Aplicação

Armazena dados frequentemente acessados em uma memória de acesso rápido, como a RAM

O cache de dados de aplicação é uma técnica que envolve armazenar temporariamente dados frequentemente acessados em uma aplicação em uma área de memória de acesso rápido, para melhorar o desempenho e reduzir a carga nos recursos do sistema

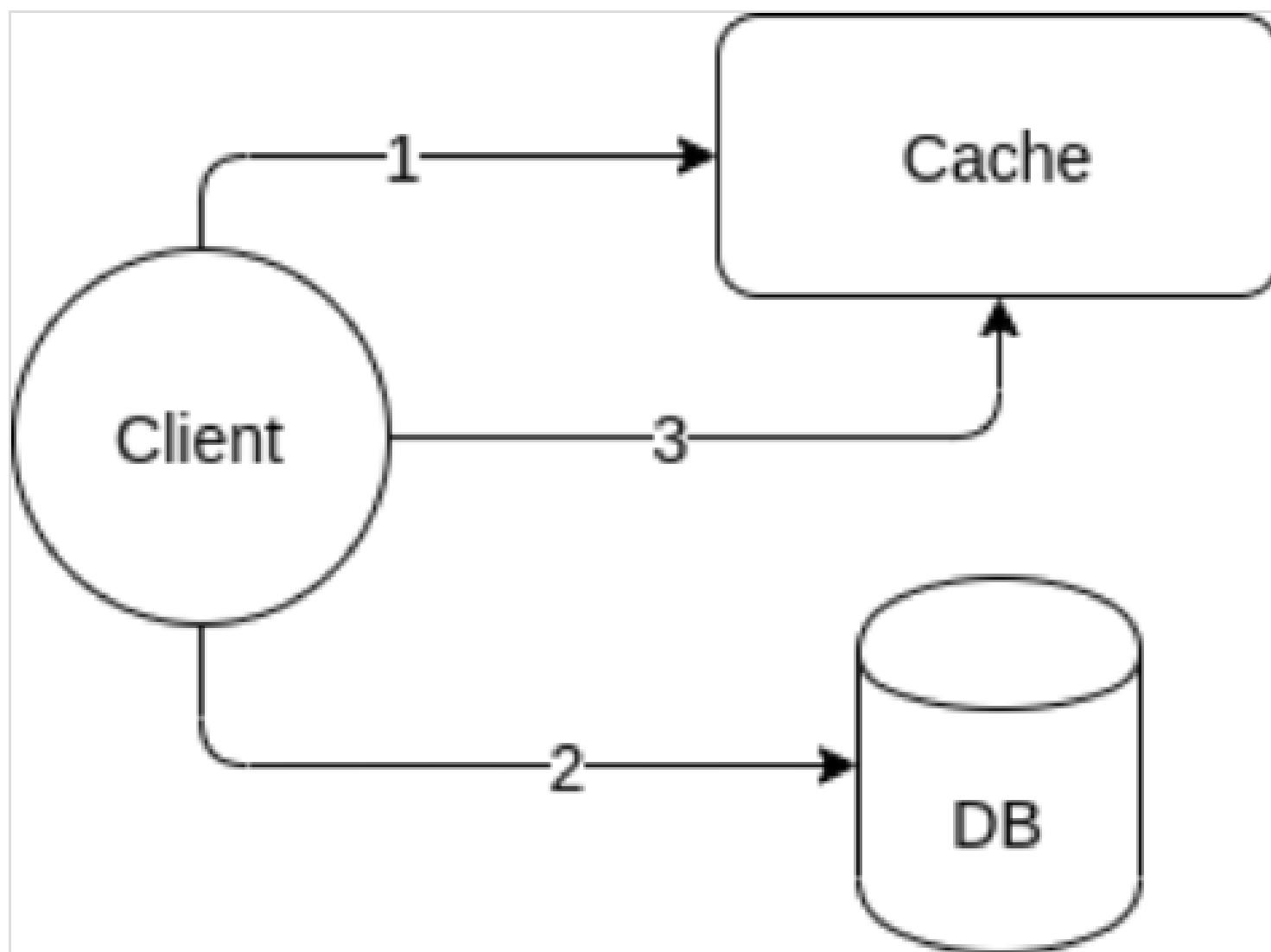
Pontos a se considerar

- Complexidade de Implementação
- Pode introduzir o risco de inconsistência de dados (diferentes partes da aplicação veem versões diferentes dos mesmos dados)
- Consumo de quantidade significativa de memória do servidor



Estratégias de atualização de cache

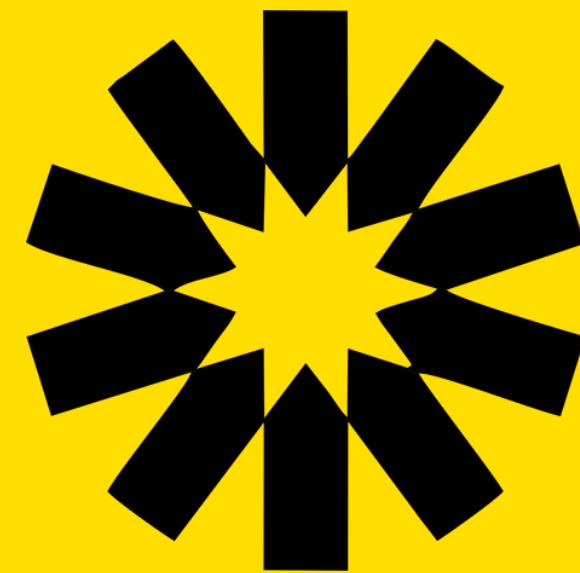
Cache-Aside



- (1) Cliente procura o dado no cache.
Não tendo, gera um “cache miss”.
- (2) Cliente busca o dado do DB.
- (3) Cliente adiciona o dado ao cache.



JSON Web Token (JWT)



JSON Web Token (JWT) é um padrão que define uma maneira compacta e independente de transmitir informações entre as partes de forma segura como um objeto JSON.



Estrutura do JWT

Header

- tipo de token
- algoritmo

hhhh.hhhh.sssss

Payload

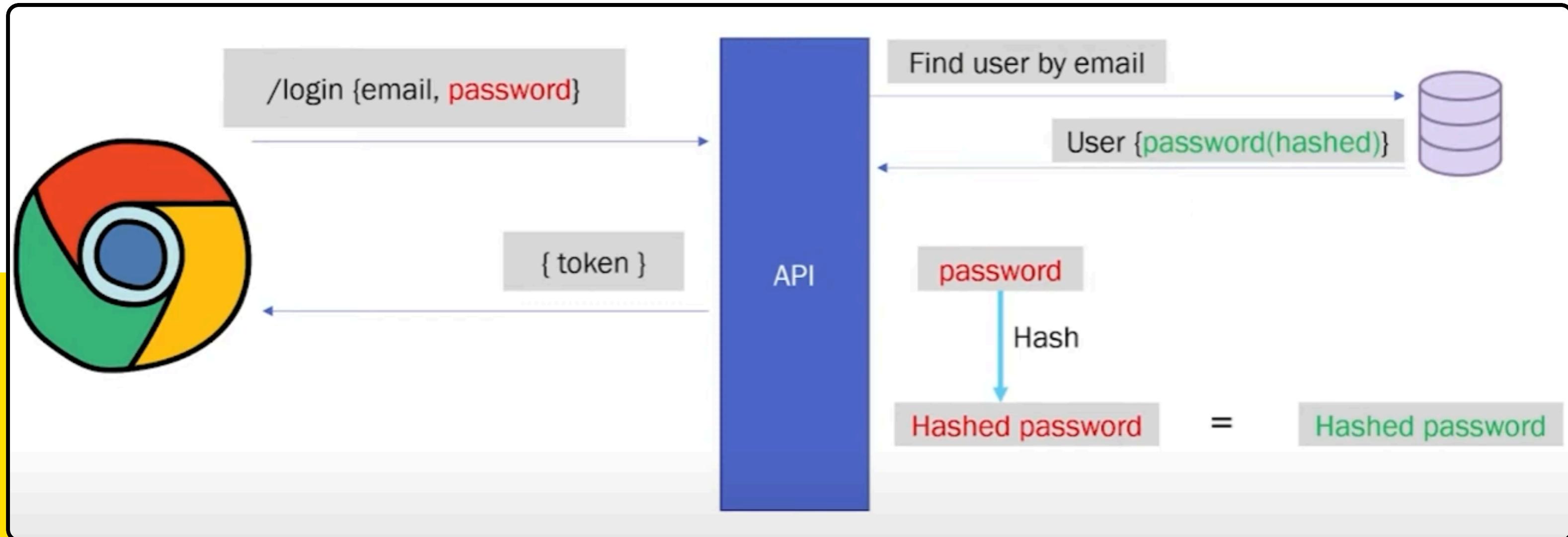
- claims
- registered/
public/
private

Signature

- base64 url
- header +
payload +
secret



Fluxo de login com JWT



Fluxo de login com JWT (Código)



```
1 @app.post("/login")
2 async def login(user: User):
3     user_found = users.find_one({"email": user.email})
4
5     if not user_found:
6         return {"message": "Usuário não encontrado", "status": "error"}
7
8     if bcrypt.checkpw(user.password.encode("utf8"), user_found["password"]):
9         jwt_token = jwt.encode({"email": user.email, "user_id": str(user_found["_id"])}, JWT_SECRET, algorithm="HS256")
10        return {"message": "Usuário logado", "token": jwt_token, "status": "success"}
11
12    return {"message": "Senha incorreta", "status": "error"}
```



Banco de dados para persistência de dados

Durabilidade

Armazenar em um meio de armazenamento permanente, como um disco rígido ou SSD.

Acesso e Recuperação

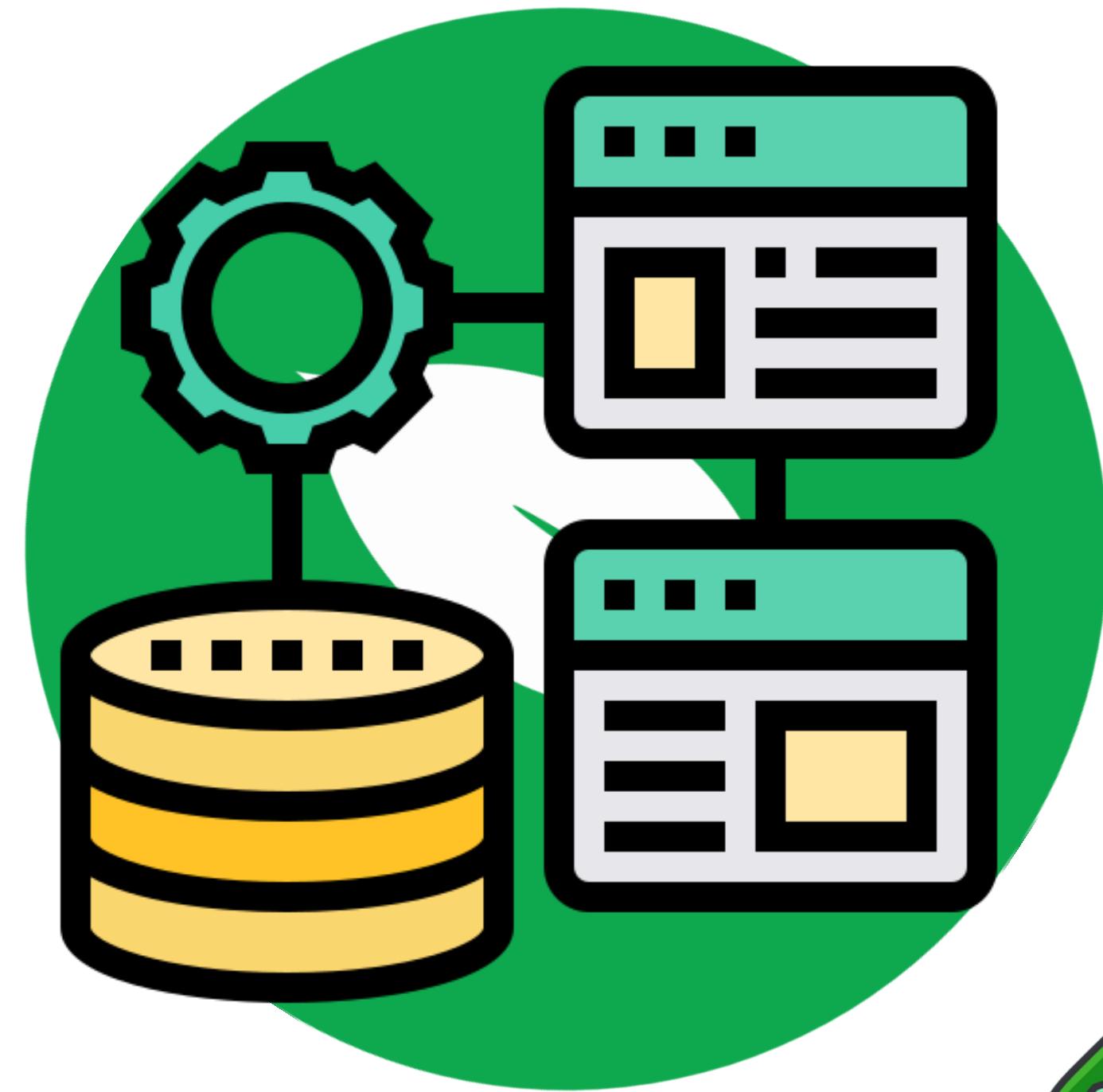
Dados persistentes devem ser acessados e recuperados a qualquer momento

Confiabilidade

Sistemas confiáveis exigem que os dados estejam disponíveis e intactos, mesmo após falhas

Consistência

Garantia de que todas as transações em um banco de dados são aplicadas de forma completa e correta



Integração do MongoDB com FastAPI

Biblioteca para interagir com o MongoDB

Connection String

Definindo o banco de dados que a aplicação utilizará

Definindo as collections

```
1 import pymongo  
2 client = pymongo.MongoClient("mongodb://localhost:27017/")  
3 database = client["pokemap"]  
4  
5 users = database["users"]  
6 pokemons_locations = database["pokemons_locations"]
```



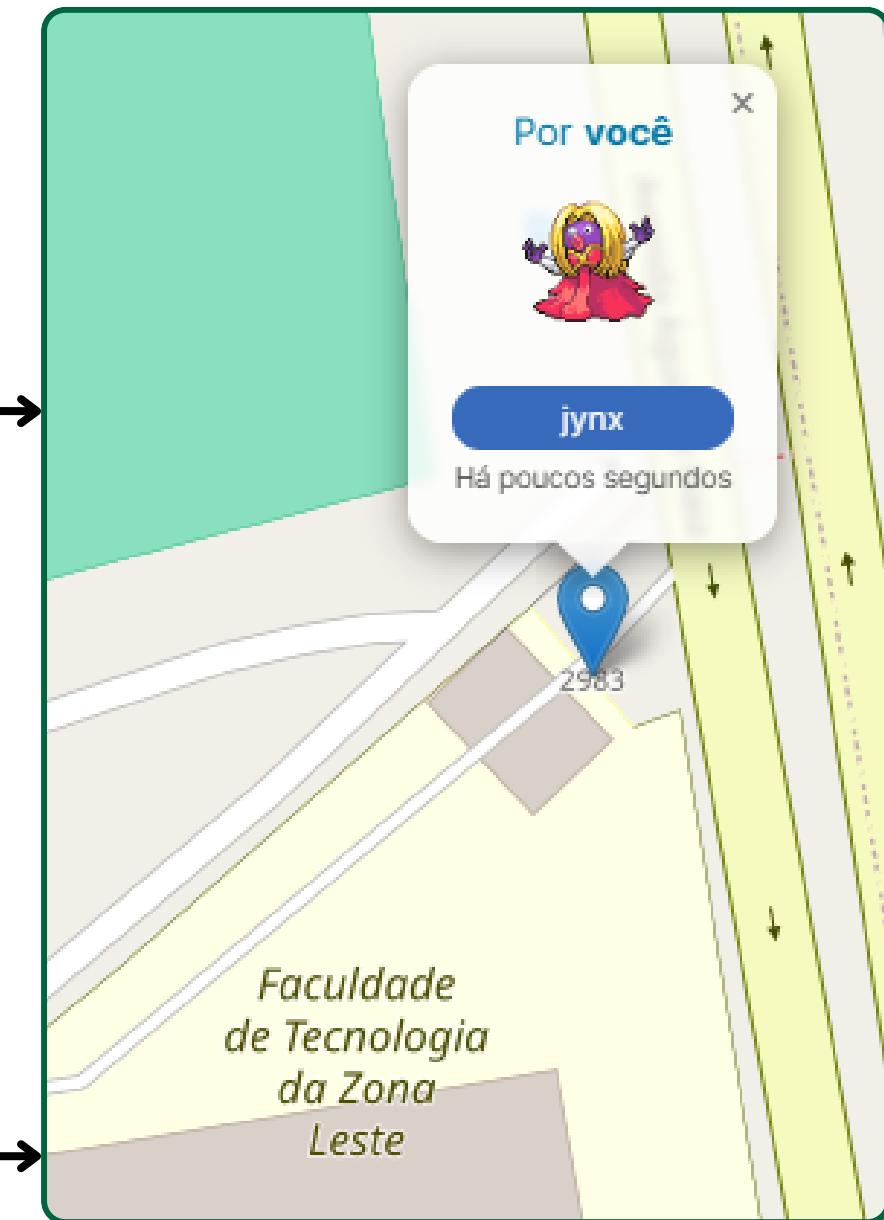
Armazenamento dos dados

```
_id: ObjectId('665fd4e0fa0a6262b227d006')
user_id : "665fd4c6fa0a6262b227d005"
pokemon_id : "124"
cep : "03694-000"
numero : "2983"
latitude : "-23.5210192"
longitude : "-46.4757933"
datetime : 2024-06-05T00:00:00:48.951+00:00
```

```
● ● ●
1 class PokemonLocation(BaseModel):
2     user_token: str
3     pokemon_id: str
4     logradouro: str
5     cep: str
6     numero: str
7     cidade: str
8     uf: str
```

```
_id: ObjectId('665fd4c6fa0a6262b227d005')
email : "teste@teste.com"
password : Binary.createFromBase64('JDJiJDEyJ')
```

```
● ● ●
1 class User(BaseModel):
2     email: str
3     password: str
```



APLICAÇÃO



```
1 class User(BaseModel):  
2     email: str  
3     password: str
```



```
1  async def register(user: User):
2      user_found = users.find_one({"email": user.email})
3      if user_found: return {"message": "Usuário já cadastrado", "status": "error"}
4
5      hashed_pass = bcrypt.hashpw(user.password.encode("utf8"), bcrypt.gensalt())
6      user_dict = {"email": user.email, "password": hashed_pass}
7      users.insert_one(user_dict)
8
9      return {"message": "Usuário criado com sucesso"}
```



```
1 @app.post("/login")
2 async def login(user: User):
3     user_found = users.find_one({"email": user.email})
4
5     if not user_found:
6         return {"message": "Usuário não encontrado", "status": "error"}
7
8     if bcrypt.checkpw(user.password.encode("utf8"), user_found["password"]):
9         jwt_token = jwt.encode({"email": user.email, "user_id": str(user_found["_id"])}, JWT_SECRET, algorithm="HS256")
10        return {"message": "Usuário logado", "token": jwt_token, "status": "success"}
11
12    return {"message": "Senha incorreta", "status": "error"}  
13
```



```
1 OPEN_CAGE_kEY = "d2a0a2ad9eff46448e031fff773fd5ec"
2 class PokemonLocation(BaseModel):
3     user_token: str
4     pokemon_id: str
5     logradouro: str
6     cep: str
7     numero: str
8     cidade: str
9     uf: str
```



```
1 @app.post("/location")
2 async def insert_location(l: PokemonLocation):
3     geocoder = OpenCageGeocode(OPEN_CAGE_kEY)
4     query = f"{l.logradouro}, {l.numero} - {l.cidade} - {l.uf}"
5     results = geocoder.geocode(query)
6     longitude = str(results[0]['geometry']['lng'])
7     latitude = str(results[0]['geometry']['lat'])
8
9     user_email = jwt.decode(l.user_token, JWT_SECRET, algorithms="HS256")["email"]
10
11    user_id = users.find_one({"email": user_email})["_id"]
12
13    p_l_dict = {
14        "user_id": str(user_id),
15        "pokemon_id": l.pokemon_id,
16        "cep": l.cep,
17        "numero": l.numero,
18        "latitude": latitude,
19        "longitude": longitude,
20        "datetime": datetime.now()
21    }
22
23    pokemons_locations.insert_one(p_l_dict)
24    return {"message": "Report inserido", "status": "success"}
```



```
1 @app.put("/location/{id}")
2     async def update_location(id, l: PokemonLocation):
3         geocoder = OpenCageGeocode(OPEN_CAGE_kEY)
4         query = f"{l.logradouro}, {l.numero} - {l.cidade} - {l.uf}"
5         results = geocoder.geocode(query)
6         longitude = str(results[0]['geometry']['lng'])
7         latitude = str(results[0]['geometry']['lat'])
8         filter = { "_id": ObjectId(id) }
9         user_email = jwt.decode(l.user_token, JWT_SECRET, algorithms="HS256")["email"]
10        user_id = users.find_one({"email": user_email})["_id"]
11        new_values = { "$set": {
12            "user_id": str(user_id),
13            "cep": l.cep,
14            "pokemon_id": l.pokemon_id,
15            "latitude": latitude,
16            "longitude": longitude
17        }}
18
19        pokemons_locations.update_one(filter, new_values)
20        return {"message": "Pokemon atualizado", "status": "success"}
```



```
1 @app.get("/locations")
2 async def get_all_locations():
3     result = pokemons_locations.find({}).sort("datetime", 1)
4     out = []
5     for location in result:
6         location["_id"] = str(location["_id"])
7         out.append(location)
8     return out
```



```
1 @app.get("/location/{id}")
2 async def update_location(id):
3     filter = { "_id": ObjectId(id) }
4     result = pokemons_locations.find_one(filter, {"_id": 0})
5     return result
```

Conclusão