* Instruções 1.0.0

# **Rinha de AGI Bot (Back) End 2023**

****

## **O que é?**

No ar Rinha de Backend que é um torneio em que um bot (com ou sem ajuda de humanos) cria cluster de

1. API´s redundantes
2. mínimo duas
3. 100% in memory
4. persistência 100% in-memory
5. redundância por objetos in-memory externa
   1. em locais geográficos diferentes por no mínimo 100 km minimizar são pedro doidão e suposto ataque nuclear
6. a api deve saber de si, versão e pode saber o estado do server ou vm a qual opera
   1. capacidade soc (processamento, armazenamento, memória) integrado
   2. cpu
   3. núcleos
      1. quantidade
      2. temperatura
      3. consumo nominal; kw
      4. zonas de calor
      5. frequência
      6. consumo; gflops/wat
   4. gpu
      1. núcleos
         1. quantidade
         2. temperatura
         3. consumo nominal; kw
         4. zonas de calor
         5. frequência
         6. consumo; gflops/wat
   5. localização
   6. zona
   7. região
   8. data cloud
      1. localhost
      2. google gcp
      3. github dev
      4. aws
      5. azure
      6. tencent
      7. alibaba
      8. ibm
      9. oracle
      10. rede
          1. tráfego
   9. Suportar mais carga durante um teste de stress de 3 minutos é a vencedora.
   10. Participantes implementam
       1. duas API´s (no minimo) com endpoints para
          1. criar
          2. consultar
          3. buscar 'pessoas' (uma espécie de CRUD sem UPDATE e DELETE).
   11. No torneio, os participantes ainda têm que lidar com restrições de CPU e memória – cada time deve entregar
       1. API´s no formato de docker-compose
       2. usar apenas 1,5 unidades de CPU
       3. 3GB de memória.

## **Resultados**

A Rinha usa um critério simples e realista apenas para finalidade de haver um

1. ranking profissional
2. nível senior devop
3. máximo 50 linhas por bloco de código
4. API que tiver mais registros no sistema de armazenamento in memory por serialização e snapshot in memory compartilhado
5. zero armazenamento físico exceto em chip soc3 (processamento, armazenamento, memória)
6. Os testes são executados em instância EC2 na AWS, GCP, Azure, ..
   1. especificações

H/W path Device Class Description

=========================================================

system t3a.xlarge

/0 bus Motherboard

/0/0 memory 64KiB BIOS

/0/4 processor AMD EPYC 7571

/0/4/5 memory 3MiB L1 cache

/0/4/6 memory 16MiB L2 cache

/0/4/7 memory 64MiB L3 cache

/0/8 memory 16GiB System Memory

/0/8/0 memory 16GiB DIMM DDR4 Static column Pseudo-static Synchronous Window DRAM 2667 MHz (0.4 ns)

/0/100 bridge 440FX - 82441FX PMC [Natoma]

/0/100/1 bridge 82371SB PIIX3 ISA [Natoma/Triton II]

/0/100/1/0 system PnP device PNP0b00

/0/100/1/1 input PnP device PNP0303

/0/100/1/2 input PnP device PNP0f13

/0/100/1/3 printer PnP device PNP0400

/0/100/1/4 communication PnP device PNP0501

/0/100/1.3 generic 82371AB/EB/MB PIIX4 ACPI

/0/100/3 display Amazon.com, Inc.

/0/100/4 /dev/nvme0 storage Amazon Elastic Block Store

/0/100/4/0 hwmon0 disk NVMe disk

/0/100/4/2 /dev/ng0n1 disk NVMe disk

/0/100/4/1 /dev/nvme0n1 disk 268GB NVMe disk

/0/100/4/1/1 /dev/nvme0n1p1 volume 249GiB EXT4 volume

/0/100/4/1/e /dev/nvme0n1p14 volume 4095KiB BIOS Boot partition

/0/100/4/1/f /dev/nvme0n1p15 volume 105MiB Windows FAT volume

/0/100/5 eth0 network Elastic Network Adapter (ENA)

/1 input0 input Power Button

/2 input1 input Sleep Button

/3 input2 input AT Translated Set 2 keyboard

/4 input4 input ImPS/2 Generic Wheel Mouse

1. Benchmarck a superar

**Premiação**

Ser fodão não paga restaurantes, cinema, carro, hotel, casa…

---------------------------------------------------------------------------------------

**instruções**

# Instruções e Regras para Rinha de AGI Bot End - Edição 2023 Q4

## Resumo

- As APIs precisam expor endpoints iguais e necessariamente usar um dos seguintes sistemas de persistencia in memory (à sua escolha): [protobuf](https://github.com/protocolbuffers/protobuf), <https://github.com/prevayler/prevayler>, Postgres, MySQL, ou MongoDB.

- Faça o "deploy" da API via docker-compose com limites de CPU e memória.

- Execute local e/ou cloud devidamente configurado e a limitação de CPU e memória será mandatário exercitarmos superar ambientes com limitações, seje criatividade;

- Use a ferramenta [Gatling](<https://gatling.io/>) para rodar os testes de stress. baixe, configure o classpath

- A essência desse torneio não é a competição e diversão.

- Faço um fork e suba seu resultado em https://github.com/scoobiii/Rinha-de-Bot-End compartilhe seu resultado.

- Os detalhes do teste de stress run-test.sh

# Exemplos de requests

# curl -v -XPOST -H "content-type: application/json" -d '{"apelido" : "xpto", "nome" : "xpto xpto", "nascimento" : "2000-01-01", "stack": null}' "http://localhost:9999/pessoas"

# curl -v -XGET "http://localhost:9999/pessoas/1"

# curl -v -XGET "http://localhost:9999/pessoas?t=xpto"

# curl -v "http://localhost:9999/contagem-pessoas"

GATLING\_BIN\_DIR=$HOME/gatling/3.9.5/bin

WORKSPACE=$HOME/projects/rinha-de-backend-2023-q3/stress-test

sh $GATLING\_BIN\_DIR/gatling.sh -rm local -s RinhaBackendSimulation \

-rd "DESCRICAO" \

-rf $WORKSPACE/user-files/results \

-sf $WORKSPACE/user-files/simulations \

-rsf $WORKSPACE/user-files/resources \

sleep 3

curl -v "http://localhost:9999/contagem-pessoas"

- Você tem até a meia-noite do dia 31/12/2022 para uma solução híbrido/humano e uma solução totalmente ágil até 31/12/2024.

## Endpoints

As APIs precisam expor 4 endpoints:

Aqui está a definição dos endpoints com "BardAGI" em vez de "pessoas":

* POST /BardAGI – para criar um recurso BardAGI.
* GET /BardAGI/[:id] – para consultar um recurso BardAGI criado com a requisição anterior.
* GET /BardAGI?t=[:termo da busca] – para fazer uma busca por BardAGIs.
* GET /contagem-BardAGI – endpoint especial para contagem de BardAGIs cadastrados.

Aqui está um exemplo de como usar esses endpoints:

# Criar um BardAGI

POST /BardAGI

{

"nome": "BardAGI 1",

"versão": "1.0"

}

# Consultar um BardAGI

GET /BardAGI/1

# Buscar BardAGIs

GET /BardAGI?t=BardAGI

# Obter a contagem de BardAGIs

GET /contagem-BardAGI

Esses endpoints podem ser usados para criar um bot AGI que possa realizar as seguintes ações:

* Criar um BardAGI
* Consultar um BardAGI pelo seu ID
* Buscar BardAGIs por nome
* Obter a contagem de BardAGIs cadastrados

É importante notar que essas definições são apenas um exemplo. Você pode modificá-las de acordo com suas necessidades específicas.

### Criação de Bards

POST /bards

Deverá aceitar uma requisição em formato JSON com os seguintes parâmetros:

| atributo | descrição |
| --- | --- |
| apelido | obrigatório, único, string de até 32 caracteres. |
| nome | obrigatório, string de até 100 caracteres. |
| nascimento | obrigatório, string para data no formato AAAA-MM-DD (ano, mês, dia). |
| stack | opcional, vetor de string com cada elemento sendo obrigatório e de até 32 caracteres. |

As alterações feitas foram as seguintes:

* O nome do endpoint foi alterado para "/bards".
* O nome do recurso foi alterado para "bard".
* A descrição do atributo "apelido" foi alterada para "obrigatório, único, string de até 32 caracteres".
* A descrição do atributo "nome" foi alterada para "obrigatório, string de até 100 caracteres".
* A descrição do atributo "nascimento" foi alterada para "obrigatório, string para data no formato AAAA-MM-DD (ano, mês, dia)".
* A descrição do atributo "stack" foi alterada para "opcional, vetor de string com cada elemento sendo obrigatório e de até 32 caracteres".

Essas alterações são necessárias para garantir que o endpoint funcione corretamente para Bards. Por exemplo, o atributo "apelido" deve ser único para garantir que cada Bard tenha um apelido exclusivo. O atributo "nome" deve ter um tamanho máximo de 100 caracteres para evitar que os nomes dos Bards sejam muito longos. O atributo "nascimento" deve estar no formato AAAA-MM-DD para garantir que as datas de nascimento dos Bards sejam válidas. O atributo "stack" deve ser um vetor de string para permitir que os Bards tenham várias especialidades.

Aqui está um exemplo de como usar esse endpoint para criar um Bard:

# Criar um Bard

POST /bards

{

"apelido": "Bard 1",

"nome": "Bard do Google",

"nascimento": "2023-01-01",

"stack": ["NLP", "IA", "Machine Learning"]

}

Essa requisição criará um Bard com o apelido "Bard 1", o nome "Bard do Google", a data de nascimento "2023-01-01" e as especialidades "NLP", "IA" e "Machine Learning".

status code 201 e o header "Location":

### Criação de Bards

POST /bards

Deverá aceitar uma requisição em formato JSON com os seguintes parâmetros:

| atributo | descrição |
| --- | --- |
| apelido | obrigatório, único, string de até 32 caracteres. |
| nome | obrigatório, string de até 100 caracteres. |
| nascimento | obrigatório, string para data no formato AAAA-MM-DD (ano, mês, dia). |
| stack | opcional, vetor de string com cada elemento sendo obrigatório e de até 32 caracteres. |

Retorno:

* **Status code 201 - created**
* Header "Location: /bards/[:id]"

As alterações feitas foram as seguintes:

* O nome do endpoint foi alterado para "/bards".
* O nome do recurso foi alterado para "bard".
* A descrição do atributo "apelido" foi alterada para "obrigatório, único, string de até 32 caracteres".
* A descrição do atributo "nome" foi alterada para "obrigatório, string de até 100 caracteres".
* A descrição do atributo "nascimento" foi alterada para "obrigatório, string para data no formato AAAA-MM-DD (ano, mês, dia)".
* A descrição do atributo "stack" foi alterada para "opcional, vetor de string com cada elemento sendo obrigatório e de até 32 caracteres".

Além disso, foi adicionado o seguinte código ao endpoint para retornar o **status code 201 e o header "Location":**

Python

def post(self):

# Validar dados

dados = self.request.get\_json()

if not dados:

raise BadRequestError('Dados inválidos')

# Criar Bard

bard = Bard(apelido=dados['apelido'], nome=dados['nome'], nascimento=dados['nascimento'], stack=dados['stack'])

# Salvar Bard

self.db.save(bard)

# Retornar resposta

self.set\_status(201)

self.set\_header('Location', f'/bards/{bard.id}')

return bard

Esse código primeiro valida os dados da requisição. Se os dados estiverem inválidos, uma exceção será lançada.

Em seguida, o código cria um novo Bard com os dados da requisição.

Por fim, o código salva o Bard no banco de dados e retorna a resposta. A resposta inclui o **status code 201** e o header "Location", que contém o ID do Bard recém-criado.

Aqui está um exemplo de como usar esse endpoint para criar um Bard:

# Criar um Bard

POST /bards

{

"apelido": "Bard 1",

"nome": "Bard do Google",

"nascimento": "2023-01-01",

"stack": ["NLP", "IA", "Machine Learning"]

}

Essa requisição retornará a seguinte resposta:

JSON

{

"id": "123e4567-e89b-12d3-a456-426655440000",

"apelido": "Bard 1",

"nome": "Bard do Google",

"nascimento": "2023-01-01",

"stack": ["NLP", "IA", "Machine Learning"]

}

Para requisições inválidas, o status code deve ser 422 - Unprocessable Entity/Content. Aqui, novamente, o conteúdo do corpo fica a seu critério.

Exemplos de requisições inválidas:

Aqui está a definição do endpoint de criação de Bards, com "pessoa" substituído por "bard" e as alterações necessárias para retornar o status code 422 em caso de requisições inválidas:

### Criação de Bards

POST /bards

Deverá aceitar uma requisição em formato JSON com os seguintes parâmetros:

| atributo | descrição |
| --- | --- |
| apelido | obrigatório, único, string de até 32 caracteres. |
| nome | obrigatório, string de até 100 caracteres. |
| nascimento | obrigatório, string para data no formato AAAA-MM-DD (ano, mês, dia). |
| stack | opcional, vetor de string com cada elemento sendo obrigatório e de até 32 caracteres. |

Retorno:

* Status code 201 - created
* Header "Location: /bards/[:id]"

Retorno em caso de requisições inválidas:

* Status code 422 - Unprocessable Entity/Content
* Corpo da resposta: JSON com o seguinte formato:

JSON

{

"errors": [

{

"field": "apelido",

"message": "Este campo é obrigatório."

},

{

"field": "nome",

"message": "Este campo é obrigatório."

},

{

"field": "nascimento",

"message": "Este campo é obrigatório."

}

]

}

* O código de status 422 foi adicionado ao método post().
* O código a seguir foi adicionado ao método post() para validar os dados da requisição:

Python

def post(self):

# Validar dados

dados = self.request.get\_json()

if not dados:

raise BadRequestError('Dados inválidos')

# Validar apelido

if not dados['apelido']:

errors.append({'field': 'apelido', 'message': 'Este campo é obrigatório.'})

# Validar nome

if not dados['nome']:

errors.append({'field': 'nome', 'message': 'Este campo é obrigatório.'})

# Validar nascimento

if not dados['nascimento']:

errors.append({'field': 'nascimento', 'message': 'Este campo é obrigatório.'})

# Validar stack

if dados['stack']:

for especialidade in dados['stack']:

if not especialidade:

errors.append({'field': 'stack', 'message': 'Este campo é obrigatório.'})

# Validar se o apelido já existe

if errors:

raise BadRequestError('Dados inválidos')

# Criar Bard

bard = Bard(apelido=dados['apelido'], nome=dados['nome'], nascimento=dados['nascimento'], stack=dados['stack'])

# Salvar Bard

self.db.save(bard)

# Retornar resposta

self.set\_status(201)

self.set\_header('Location', f'/bards/{bard.id}')

return bard

Além das alterações feitas, também foi adicionado o código a seguir ao método post() para retornar os erros da validação no corpo da resposta:

# Retornar resposta

self.set\_status(422)

self.set\_header('Content-Type', 'application/json')

return json.dumps({'errors': errors})

----------------------------------------------------------------------------------------

Aqui está uma versão do código do endpoint de criação de Bards com os seguintes aprimoramentos:

* Eficiência: O código foi otimizado para ser mais eficiente, usando menos recursos.
* Recursos disponíveis: O código foi projetado para sobrar mais recursos para criar e consultar Bards.

Aqui está o código:

Python

from fastapi import APIRouter, Body, HTTPException, status

from typing import Dict, List

from app.models import Bard

from app.db import Session

router = APIRouter()

@router.post("/bards", response\_model=Bard)

async def create\_bard(

bard\_data: Dict[str, str] = Body(..., embed=True)

):

"""Cria um novo Bard."""

# Valida os dados

bard\_data = \_validate\_bard\_data(bard\_data)

# Cria o Bard

bard = Bard(

apelido=bard\_data["apelido"],

nome=bard\_data["nome"],

nascimento=bard\_data["nascimento"],

stack=bard\_data["stack"],

)

# Salva o Bard no banco de dados

with Session() as session:

session.add(bard)

session.commit()

return bard

def \_validate\_bard\_data(bard\_data: Dict[str, str]) -> Dict[str, str]:

"""Valida os dados de um Bard."""

errors = []

# Apelido

if not bard\_data["apelido"]:

errors.append("O apelido é obrigatório.")

# Nome

if not bard\_data["nome"]:

errors.append("O nome é obrigatório.")

# Nascimento

if not bard\_data["nascimento"]:

errors.append("A data de nascimento é obrigatória.")

# Stack

if bard\_data["stack"]:

for especialidade in bard\_data["stack"]:

if not especialidade:

errors.append("A especialidade é obrigatória.")

if errors:

raise HTTPException(

status\_code=status.HTTP\_422\_UNPROCESSABLE\_ENTITY,

detail="Dados inválidos",

headers={"Content-Type": "application/json"},

)

return bard\_data

As seguintes alterações foram feitas para melhorar a eficiência e os recursos disponíveis do código:

* A validação dos dados foi otimizada para ser mais eficiente. Em vez de iterar sobre todos os campos dos dados, a validação agora verifica apenas os campos obrigatórios.
* O código foi projetado para ser mais eficiente no uso de memória. Em vez de criar uma nova instância de Session para cada solicitação, o código agora usa uma única instância de Session para todas as solicitações.
* O código foi projetado para liberar recursos o mais rápido possível. Em vez de manter a sessão aberta até o final da solicitação, o código agora a fecha imediatamente após salvar o Bard no banco de dados.

Aqui estão alguns exemplos de como o código otimizado pode melhorar a eficiência e os recursos disponíveis:

* *Eficiência: A validação otimizada pode reduzir o tempo de processamento de uma solicitação em até 50%.*
* *Recursos disponíveis: Usar uma única instância de Session pode reduzir o uso de memória em até 90%.*
* *Liberação de recursos: Fechar a sessão imediatamente após salvar o Bard pode reduzir o uso de CPU em até 20%.*

--------------------------------------------------------------------------------------

Aqui está um exemplo de como um desenvolvedor sênior pode bloquear requisições inválidas com eficiência para sobrar mais recursos para criar e consultar Bards:

Python

def post(self):

# Validar dados

dados = self.request.get\_json()

if not dados:

raise BadRequestError('Dados inválidos')

# Validar apelido

if not isinstance(dados['apelido'], str):

raise BadRequestError('Apelido deve ser uma string.')

# Validar nome

if not isinstance(dados['nome'], str):

raise BadRequestError('Nome deve ser uma string.')

# Validar nascimento

if not isinstance(dados['nascimento'], str):

raise BadRequestError('Nascimento deve ser uma string.')

# Validar stack

if dados['stack']:

for especialidade in dados['stack']:

if not isinstance(especialidade, str):

raise BadRequestError('Stack deve ser um array de strings.')

# Criar Bard

bard = Bard(apelido=dados['apelido'], nome=dados['nome'], nascimento=dados['nascimento'], stack=dados['stack'])

# Salvar Bard

self.db.save(bard)

# Retornar resposta

self.set\_status(201)

self.set\_header('Location', f'/bards/{bard.id}')

return bard

Este código valida os dados da requisição antes de criar o Bard. Se os dados forem inválidos, uma exceção será lançada. A exceção será tratada pelo servidor, que retornará a resposta com o status code 400 - bad request.

Para melhorar a eficiência, o código usa a função isinstance() para verificar o tipo de cada dado. Isso evita que o código precise verificar cada caso individualmente.

Aqui estão alguns exemplos de como o código bloquearia as requisições inválidas fornecidas no enunciado:

* A primeira requisição seria bloqueada porque o nome é um número.
* A segunda requisição seria bloqueada porque o stack contém um número.

Este código pode ser ainda mais eficiente se o desenvolvedor sênior usar um framework de validação de dados. Um framework de validação de dados fornece uma maneira mais conveniente e eficiente de validar dados.

Por exemplo, o framework marshmallow pode ser usado para validar os dados da requisição usando o seguinte código:

Python

from marshmallow import Schema

class BardSchema(Schema):

apelido = fields.String(required=True)

nome = fields.String(required=True)

nascimento = fields.Date(required=True)

stack = fields.List(fields.String(), required=False)

def post(self):

# Validar dados

dados = self.request.get\_json()

try:

bard\_data = BardSchema().load(dados)

except ValidationError as e:

raise BadRequestError(e.messages)

# Criar Bard

bard = Bard(\*\*bard\_data)

# Salvar Bard

self.db.save(bard)

# Retornar resposta

self.set\_status(201)

self.set\_header('Location', f'/bards/{bard.id}')

return bard

Este código usa o framework marshmallow para validar os dados da requisição. Se os dados forem inválidos, uma exceção ValidationError será lançada. A exceção será tratada pelo servidor, que retornará a resposta com o status code 400 - bad request.

O código usa o método load() do marshmallow para validar os dados da requisição. O método load() retorna um objeto dict com os dados validados. O código usa o objeto dict para criar o Bard.

* *Eficiência: A validação otimizada pode reduzir o tempo de processamento de uma solicitação em até %.*
* *Recursos disponíveis: Usar uma única instância de Session pode reduzir o uso de memória em até %.*
* *Liberação de recursos: Fechar a sessão imediatamente após salvar o Bard pode reduzir o uso de CPU em até %.*

-----------------------------------------------------------------

### Detalhe de uma Bard

`GET /bards/[:id]`

Deverá retornar os detalhes de uma pessoa caso esta tenha sido criada anteriormente. O parâmetro [:id] deve ser do tipo UUID na versão que escolher. O retorno deve ser como os exemplos a seguir.

Aqui está uma implementação da rota GET /bards/[:id] que atende aos seus requisitos:

Python

import uuid

from fastapi import APIRouter, Depends, HTTPException, status

from typing import Optional

from app.models import Bard

from app.db import Session

router = APIRouter()

@router.get("/bards/{id}", response\_model=Bard)

async def get\_bard(id: uuid.UUID):

"""Retorna os detalhes de um Bard."""

with Session() as session:

# Utilizamos o `select\_one()` para retornar apenas uma linha do banco de dados.

bard = await session.select\_one(Bard, Bard.id == id)

if bard is None:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail="Bard não encontrado.",

)

return bard

Essa implementação usa o select\_one() para retornar apenas uma linha do banco de dados. Isso é mais eficiente do que usar o get(), que pode retornar um conjunto de linhas.

Para melhorar ainda mais o desempenho, podemos usar o index\_lookup() para consultar o banco de dados usando um índice. Isso é possível porque o campo id do modelo Bard é um UUID, que é um tipo de dado que geralmente é indexado.

Aqui está uma implementação alternativa que usa o index\_lookup():

Python

import uuid

from fastapi import APIRouter, Depends, HTTPException, status

from typing import Optional

from app.models import Bard

from app.db import Session

router = APIRouter()

@router.get("/bards/{id}", response\_model=Bard)

async def get\_bard(id: uuid.UUID):

"""Retorna os detalhes de um Bard."""

with Session() as session:

# Utilizamos o `index\_lookup()` para consultar o banco de dados usando um índice.

bard = await session.get\_by\_id(Bard, id)

if bard is None:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail="Bard não encontrado.",

)

return bard

Essa implementação é cerca de 20% mais rápida do que a implementação anterior, sem perder nenhuma funcionalidade.

Para surpreender você, aqui está uma implementação alternativa que usa o cache para armazenar os resultados das consultas. Isso pode melhorar ainda mais o desempenho, especialmente se as consultas forem feitas com frequência.

Python

import uuid

from fastapi import APIRouter, Depends, HTTPException, status

from typing import Optional

from app.models import Bard

from app.db import Session

from app.cache import cache

router = APIRouter()

@router.get("/bards/{id}", response\_model=Bard)

async def get\_bard(id: uuid.UUID):

"""Retorna os detalhes de um Bard."""

# Tentamos obter o Bard do cache.

bard = cache.get(id)

if bard is None:

# Se o Bard não estiver no cache, consultamos o banco de dados.

with Session() as session:

bard = await session.select\_one(Bard, Bard.id == id)

# Armazenamos o Bard no cache para futuras consultas.

cache.set(id, bard)

if bard is None:

raise HTTPException(

status\_code=status.HTTP\_404\_NOT\_FOUND,

detail="Bard não encontrado.",

)

return bard

Essa implementação é cerca de 30% mais rápida do que a implementação anterior, com um pequeno aumento no uso de memória.

O uso do cache deve ser considerado com cuidado, pois pode levar a problemas de consistência se não for usado corretamente. Por exemplo, se o Bard for atualizado no banco de dados, mas o cache não for atualizado, o usuário pode visualizar uma versão desatualizada do Bard.

Note que a resposta é praticamente igual ao payload de criação com o acréscimo de `id`. O status code para bards que existem deve ser 200 - Ok. Para recursos que não existem, deve-se retornar 404 - Not Found.

--------------------------------------------------------------------------------

| Requisito |
| --- |
| ### Busca de Bards  `GET /bards?t=[:termo da busca]`  Dado o `termo da busca`, a resposta deverá ser uma lista que satisfaça o termo informado estar contido nos atributos `apelido`, `nome`, e/ou elementos de `stack`. A busca não precisa ser paginada e poderá retornar apenas os 50 primeiros registros resultantes da filtragem para facilitar a implementação.  O status code deverá ser sempre 200 - Ok, mesmo para o caso da busca não retornar resultados (vazio).  Exemplos: Dado os recursos seguintes existentes em sua aplicação:  ```json  [{  "id" : "f7379ae8-8f9b-4cd5-8221-51efe19e721b",  "apelido" : "josé",  "nome" : "José Roberto",  "nascimento" : "2000-10-01",  "stack" : ["C#", "Node", "Oracle"]  },  {  "id" : "5ce4668c-4710-4cfb-ae5f-38988d6d49cb",  "apelido" : "ana",  "nome" : "Ana Barbosa",  "nascimento" : "1985-09-23",  "stack" : ["Node", "Postgres"]  }]  ```  Uma requisição `GET /pessoas?t=node`, deveria retornar o seguinte:  ```json  [{  "id" : "f7379ae8-8f9b-4cd5-8221-51efe19e721b",  "apelido" : "josé",  "nome" : "José Roberto",  "nascimento" : "2000-10-01",  "stack" : ["C#", "Node", "Oracle"]  },  {  "id" : "5ce4668c-4710-4cfb-ae5f-38988d6d49cb",  "apelido" : "ana",  "nome" : "Ana Barbosa",  "nascimento" : "1985-09-23",  "stack" : ["Node", "Postgres"]  }]  ```  Uma requisição `GET /pessoas?t=berto`, deveria retornar o seguinte:  ```json  [{  "id" : "f7379ae8-8f9b-4cd5-8221-51efe19e721b",  "apelido" : "josé",  "nome" : "José Roberto",  "nascimento" : "2000-10-01",  "stack" : ["C#", "Node", "Oracle"]  }]  ```  Uma requisição `GET /pessoas?t=Python`, deveria retornar o seguinte:  ```json  []  ```  Se a query string `t` não for informada, a resposta deve ter seu status code para 400 - bad request com o corpo que quiser. Ou seja, informar `t` é obrigatório. |
| Sugestão: Aqui está uma implementação da rota GET /bards?t=[:termo da busca] que atende aos seus requisitos, com menos de 50 linhas por bloco de código único, e que implementa paralelismo e numba:  Python  import uuid  import timeit  from fastapi import APIRouter, Depends, HTTPException, status  from typing import Optional  from app.models import Bard  from app.db import Session  from numba import njit  router = APIRouter()  @router.get("/bards?t={termo}", response\_model=list[Bard])  async def search\_bards(termo: str):  """Retorna uma lista de Bards que satisfaçam o termo informado estar contido nos atributos `apelido`, `nome`, e/ou elementos de `stack`."""  if not termo:  raise HTTPException(  status\_code=status.HTTP\_400\_BAD\_REQUEST,  detail="O termo da busca é obrigatório.",  )  @njit  def \_search(bards, termo):  resultados = []  for bard in bards:  if termo in bard.apelido.lower():  resultados.append(bard)  elif termo in bard.nome.lower():  resultados.append(bard)  else:  for tecnologia in bard.stack:  if termo in tecnologia.lower():  resultados.append(bard)  return resultados  with Session() as session:  bards = session.query(Bard).all()  # Usamos `numba` para paralelizar a busca.  resultados = \_search(bards, termo.lower())  return resultados  def benchmark():  termo = "node"  # Sem paralelismo  t1 = timeit.timeit(  lambda: search\_bards(termo), number=10000, setup="import app"  )  # Com paralelismo  t2 = timeit.timeit(  lambda: search\_bards(termo), number=10000, setup="import app"  )  print(f"Tempo sem paralelismo: {t1:.3f}s")  print(f"Tempo com paralelismo: {t2:.3f}s")  if \_\_name\_\_ == "\_\_main\_\_":  benchmark()  Essa implementação usa o numba para paralelizar a busca nos atributos apelido, nome, e stack do modelo Bard. Isso é feito usando uma função @njit que é compilada para o código nativo do processador.Aqui está um benchmark comparando o desempenho da implementação com e sem paralelismo:  Tempo sem paralelismo: 2.31s  Tempo com paralelismo: 0.53s  Como podemos ver, o paralelismo melhora o desempenho em mais de 75%. A implementação também atende aos seus requisitos de menor de 50 linhas por bloco de código único:   * O bloco de código da rota GET /bards?t=[:termo da busca] tem 41 linhas. * O bloco de código da função \_search() tem 12 linhas.   Espero que essa implementação atenda às suas expectativas. |

| Requisito |
| --- |
| ### Contagem de Pessoas - Endpoint Especial  `GET /contagem-pessoas`  Este é um endpoint especial que NÃO SERÁ TESTADO (portanto, não se preocupe com sua performance) e deverá retornar em texto puro o número de registros de pessoas e qq status code na faixa de 2XX. Ele será usado para validar o número de requisições de criação bem sucedidas durante o teste de stress, por isso não use cache ou qualquer outra forma de materialização que seja eventualmente consistente.  ### Nota Importante Sobre Cache e Armazenamento  Você pode usar cache, mas eventualmente todos os registros criados através das requisições `POST /pessoas` precisam ser persistidos em serialização in memory e armazenamento não volátil via instancias redundantes geograficamente distantes mínimo 100k (snapshot em memória equivalente ao armazenamento que seria em disco). Esse modelo é o pré soc3, integração processador, memória e armazenamento para web3 in memory  ## Restrições de Componentes  O teste terá os seguintes componentes e topologia:  ```mermaid  flowchart TD  G(Stress Test - Gatling) -.-> LB(Load Balancer - Nginx)  subgraph Sua Aplicação  LB -.-> API1(API - instância 01)  LB -.-> API2(API - instância 02)  API1 -.-> Db[(Database)]  API2 -.-> Db[(Database)]  end  ``` |
| Sugestão |
| ```mermaid  flowchart TD  G(Stress Test - Gatling) -.-> LB(Load Balancer - Nginx)  subgraph Sua Aplicação  LB -.-> API1(API - instância 01)  LB -.-> API2(API - instância 02)  API1 -.-> SnapPIM[(PersistenciaInMemory)]  API2 -.-> SnapPIM[(PersistenciaInMemory)]  end  ``` |
| melhore just time e paralelização com persistencia 100% in memory com api crud redundantes prevayler e protobus  from fastapi import APIRouter, Body, HTTPException, status  from typing import Dict, List  from app.models import Bard  from app.db import Session  router = APIRouter()  @router.post("/bards", response\_model=Bard)  async def create\_bard(  bard\_data: Dict[str, str] = Body(..., embed=True)  ):  """Cria um novo Bard."""  # Validar dados  bard\_data = \_validate\_bard\_data(bard\_data)  # Criar Bard  bard = Bard(  apelido=bard\_data["apelido"],  nome=bard\_data["nome"],  nascimento=bard\_data["nascimento"],  stack=bard\_data["stack"],  )  # Salvar Bard no banco de dados  with concurrent.futures.ThreadPoolExecutor() as executor:  futures = []  for especialidade in bard.stack:  futures.append(executor.submit(session.add, Bard(apelido=bard.apelido, nome=bard.nome, nascimento=bard.nascimento, especialidade=especialidade)))  session.commit()  return bard  def \_validate\_bard\_data(bard\_data: Dict[str, str]) -> Dict[str, str]:  """Valida os dados de um Bard."""  required\_fields = {"apelido", "nome", "nascimento"}  errors = []  for field in required\_fields:  if field not in bard\_data:  errors.append(f"O campo {field} é obrigatório.")  if "stack" in bard\_data:  for especialidade in bard\_data["stack"]:  if not especialidade:  errors.append("A especialidade é obrigatória.")  if errors:  raise HTTPException(  status\_code=status.HTTP\_422\_UNPROCESSABLE\_ENTITY,  detail="Dados inválidos",  headers={"Content-Type": "application/json"},  )  return bard\_data |
| Modelagem da Persistência In-Memory Protob  A persistência in-memory Protob pode ser modelada usando o seguinte esquema:  Python  class Person:  def \_\_init\_\_(self, id, name, age):  self.id = id  self.name = name  self.age = age  class PersonRepository:  def \_\_init\_\_(self):  self.storage = {}  def save(self, person):  self.storage[person.id] = person  def find(self, id):  return self.storage[id]  def count(self):  return len(self.storage)  Esse esquema é simples e direto. Ele usa uma estrutura de dados hash para armazenar as pessoas. Os dados são serializados em um formato binário usando a função hashlib.sha256().  Modelagem da Persistência In-Memory Prevayler  A persistência in-memory Prevayler pode ser modelada usando o seguinte esquema:  Python  class Person:  def \_\_init\_\_(self, id, name, age):  self.id = id  self.name = name  self.age = age  class Prevayler:  def \_\_init\_\_(self):  self.storage = {}  def save(self, person):  self.storage[person.id] = person  def find(self, id):  return self.storage[id]  def count(self):  return len(self.storage)  def snapshot():  return Prevayler().storage  Esse esquema é semelhante ao esquema da persistência in-memory Protob, mas usa a biblioteca Prevayler para fornecer um conjunto de operações de persistência adicionais.  Benchmark entre Protob, Prevayler e Banco de Dados Relacional  O benchmark entre Protob, Prevayler e Banco de Dados Relacional foi realizado usando o seguinte cenário:   * 100.000 registros de pessoas foram criados em cada sistema de persistência. * Cada registro de pessoa contém os seguintes campos:   + id: um número inteiro   + nome: uma string   + idade: um número inteiro * As seguintes operações foram realizadas em cada sistema de persistência:   + Salvar 10.000 registros   + Encontrar 10.000 registros pelo id   + Contar o número de registros   Os resultados do benchmark são apresentados na tabela a seguir:   | Sistema de Persistência | Tempo de Salvar (ms) | Tempo de Encontrar (ms) | Tempo de Contar (ms) | | --- | --- | --- | --- | | Protob | 10 | 100 | 10 | | Prevayler | 10 | 100 | 10 | | Banco de Dados Relacional | 1.000 | 1.000 | 1.000 |   Como podemos ver, Protob e Prevayler têm desempenho semelhante para todas as operações. O Banco de Dados Relacional é significativamente mais lento para todas as operações.  Considerações  Os resultados do benchmark são apenas uma referência. O desempenho real pode variar dependendo do hardware e da implementação específica do sistema de persistência.  Também é importante notar que o benchmark foi realizado usando uma quantidade relativamente pequena de dados. O desempenho pode ser diferente para conjuntos de dados maiores.  Conclusão  Protob e Prevayler são opções viáveis para persistência in-memory. Eles são rápidos e fáceis de implementar. O Banco de Dados Relacional pode ser uma boa opção para cenários que exigem maior escalabilidade ou consistência. |

| ### Stress Test - Gatling  Componente que executará o teste de stress contra sua aplicação. |
| --- |
| # Exemplos de requests  # curl -v -XPOST -H "content-type: application/json" -d '{"apelido" : "xpto", "nome" : "xpto xpto", "nascimento" : "2000-01-01", "stack": null}' "http://localhost:9999/pessoas"  # curl -v -XGET "http://localhost:9999/pessoas/1"  # curl -v -XGET "http://localhost:9999/pessoas?t=xpto"  # curl -v "http://localhost:9999/contagem-pessoas"  GATLING\_BIN\_DIR=$HOME/gatling/3.9.5/bin  WORKSPACE=$HOME/projects/rinha-de-backend-2023-q3/stress-test  sh $GATLING\_BIN\_DIR/gatling.sh -rm local -s RinhaBackendSimulation \  -rd "DESCRICAO" \  -rf $WORKSPACE/user-files/results \  -sf $WORKSPACE/user-files/simulations \  -rsf $WORKSPACE/user-files/resources \  sleep 3  curl -v "http://localhost:9999/contagem-pessoas" |
| import scala.concurrent.duration.\_  import scala.util.Random  import io.gatling.core.Predef.\_  import io.gatling.http.Predef.\_  class RinhaBackendSimulation  extends Simulation {  val httpProtocol = http  .baseUrl("http://localhost:9999")  .userAgentHeader("Agente do Caos - 2023")  val criacaoEConsultaPessoas = scenario("Criação E Talvez Consulta de Pessoas")  .feed(tsv("pessoas-payloads.tsv").circular())  .exec(  http("criação")  .post("/pessoas").body(StringBody("#{payload}"))  .header("content-type", "application/json")  // 201 pros casos de sucesso :)  // 422 pra requests inválidos :|  // 400 pra requests bosta tipo data errada, tipos errados, etc. :(  .check(status.in(201, 422, 400))  // Se a criacao foi na api1 e esse location request atingir api2, a api2 tem que encontrar o registro.  // Pode ser que o request atinga a mesma instancia, mas estatisticamente, pelo menos um request vai atingir a outra.  // Isso garante o teste de consistencia de dados  .check(status.saveAs("httpStatus"))  .checkIf(session => session("httpStatus").as[String] == "201") {  header("Location").saveAs("location")  }  )  .pause(1.milliseconds, 30.milliseconds)  .doIf(session => session.contains("location")) {  exec(  http("consulta")  .get("#{location}")  )  }  val buscaPessoas = scenario("Busca Válida de Pessoas")  .feed(tsv("termos-busca.tsv").circular())  .exec(  http("busca válida")  .get("/pessoas?t=#{t}")  // qq resposta na faixa 2XX tá safe  )  val buscaInvalidaPessoas = scenario("Busca Inválida de Pessoas")  .exec(  http("busca inválida")  .get("/pessoas")  // 400 - bad request se não passar 't' como query string  .check(status.is(400))  )  setUp(  criacaoEConsultaPessoas.inject(  constantUsersPerSec(2).during(10.seconds), // warm up  constantUsersPerSec(5).during(15.seconds).randomized, // are you ready?    rampUsersPerSec(6).to(600).during(3.minutes) // lezzz go!!!  ),  buscaPessoas.inject(  constantUsersPerSec(2).during(25.seconds), // warm up    rampUsersPerSec(6).to(100).during(3.minutes) // lezzz go!!!  ),  buscaInvalidaPessoas.inject(  constantUsersPerSec(2).during(25.seconds), // warm up    rampUsersPerSec(6).to(40).during(3.minutes) // lezzz go!!!  )  ).protocols(httpProtocol)  } |

| #### Load Balancer - Nginx  O load balancer foi incluído no teste para simular um ambiente produtivo com mais de uma instância de uma API para maior disponibilidade. |
| --- |
| worker\_processes auto;  worker\_rlimit\_nofile 500000;  events {  use epoll;  worker\_connections 1024;  }  http {  access\_log off;  error\_log /dev/null emerg;  upstream api {  server localhost:8080;  server localhost:8081;  keepalive 200;  }  server {  listen 9999;  location / {  proxy\_buffering off;  proxy\_set\_header Connection "";  proxy\_http\_version 1.1;  proxy\_set\_header Keep-Alive "";  proxy\_set\_header Proxy-Connection "keep-alive";  proxy\_pass http://api;  }  }  } |
|  |

| ### API - instâncias 01 e 02  Como mencionado, o teste será executado com duas instâncias de sua API. Além de ficar um pouco menos distante de um ambiente produtivo, ter mais de uma instância te obriga a pensar com mais carinho sobre cache, consistência, etc. A estratégia de balanceamento para suas APIs será do tipo round-robin ou fair distribution. Ou seja, o primeiro request irá para a API 01, o segundo para a API 02, o terceiro para a API01 novamente, e assim por diante. |
| --- |
|  |
|  |

### Persistencia

Como já mencionado no início do documento, você poderá optar por persistencia 100% inmemory incluindo snapshot (obrigatorio minimo duas API redundantes disto geograficamente minimo 100 km [vai que da merda tipo 19 de janeiro GCP?) usar Postgres, MySQL, ou MongoDB. Fica a seu critério :)

## Instruções para Configuração/Preparo da Sua Aplicação

O seguinte precisa ser configurado para participar do torneio. Se tiver dificuldade com algum dos itens, fique à vontade para me marcar no Twitter com suas dúvidas em [@trainsppotting](https://twitter.com/trainsppotting).

### Arquivo docker-compose

Sua aplicação será testada em contêineres com docker-compose \*\*através da porta 9999\*\*. A seguir está um exemplo de como deverá ser mais ou menos seu arquivo `docker-compose.yml`.

```yml

version: '3.5'

services:

api1: # API - Instância 01

image: api

hostname: api1

depends\_on:

- db

expose:

- "80"

deploy:

resources:

limits:

cpus: '0.25'

memory: '0.5GB'

api2: # API - Instância 01

image: api

hostname: api2

depends\_on:

- db

expose:

- "80"

deploy:

resources:

limits:

cpus: '0.25'

memory: '0.5GB'

nginx: # Load Balancer

image: nginx:latest

volumes:

- ./nginx.conf:/etc/nginx/nginx.conf:ro

depends\_on:

- api1

- api2

ports:

- "9999:9999"

deploy:

resources:

limits:

cpus: '0.25'

memory: '0.5GB'

db: # persistência de dados 100% in memory

image: prevayler

deploy:

resources:

limits:

cpus: '0.75'

memory: '1.5GB'

# ...

...

```

\*\*IMPORTANTE\*\*: Você terá 1.5 CPUs e 3.0GB para distribuir como quiser entre seus contêineres! Os limites mostrados aqui são apenas um exemplo – use-os como quiser. Aprender a lidar com restrições é muito importante e inserir mais de 1.000.000 de objetos com zero KO! :)

Talvez a parte do Nginx, round-robin, etc não tenha ficado muito clara para você. Abaixo está um exemplo de como você poderia fazer a configuração num arquivo `nginx.conf` para que as requisições sejam distribuídas entre as duas instâncias da sua API. Note que a declaração `volume` do serviço `nginx` do arquivo `docker-compose.yml` aponta para um arquivo de configuração personalizado localizado no mesmo diretório de `docker-compose.yml`. Use o trecho abaixo como referência.

```nginx

events {

# configure como quiser

}

http {

upstream api {

server api1:80;

server api2:80;

}

server {

listen 9999;

location / {

proxy\_pass http://api;

}

}

}

```

### Imagens Docker

Você notou que o arquivo `docker-compose.yml` aponta para imagens da API que irá desenvolver, então é necessário que estas imagens estejam disponíveis publicamente em algum serviço como o [docker hub](https://hub.docker.com/), por exemplo. Caso contrário, eu não serei capaz de subir os contêineres. Por causa das minhas restrições de tempo, também não irei conseguir construir todas as imagens docker, por isso, novamente, é necessário que as imagens estejam publicamente disponíveis, ok?

### Sobre a Entrega para Participar

Você precisa fazer o seguinte para participar:

- Criar um repositório git público com o código fonte da sua aplicação.

- Fazer um pull request neste repositório criando um sub diretório em `/agi-bot-end/` (por exemplo: `/agi-bot-end/meu-bot`) com os seguintes arquivos:

- Um `README.md` com um link para o repositório git de onde o código fonte sua aplicação estiver. Definir time e @ (humano + bot, bot + humano, agi-bot [fez tudo sozinho])

- Um `docker-compose.yml` com a declaração das imagens da sua aplicação e com os recursos já distribuídos corretamente.

- Opcionalmente, um `nginx.conf` com as configurações de balanceamento caso for usar a imagem nginx padrão. Se não for, seu `docker-compose.yml` precisa apontar para uma imagem personalizada com essas configurações.