Documentação Técnica

Autor: Alexandre Otávio Guedes Drummond

Nome do Software: Clock Api

Versão do Software: 1.0

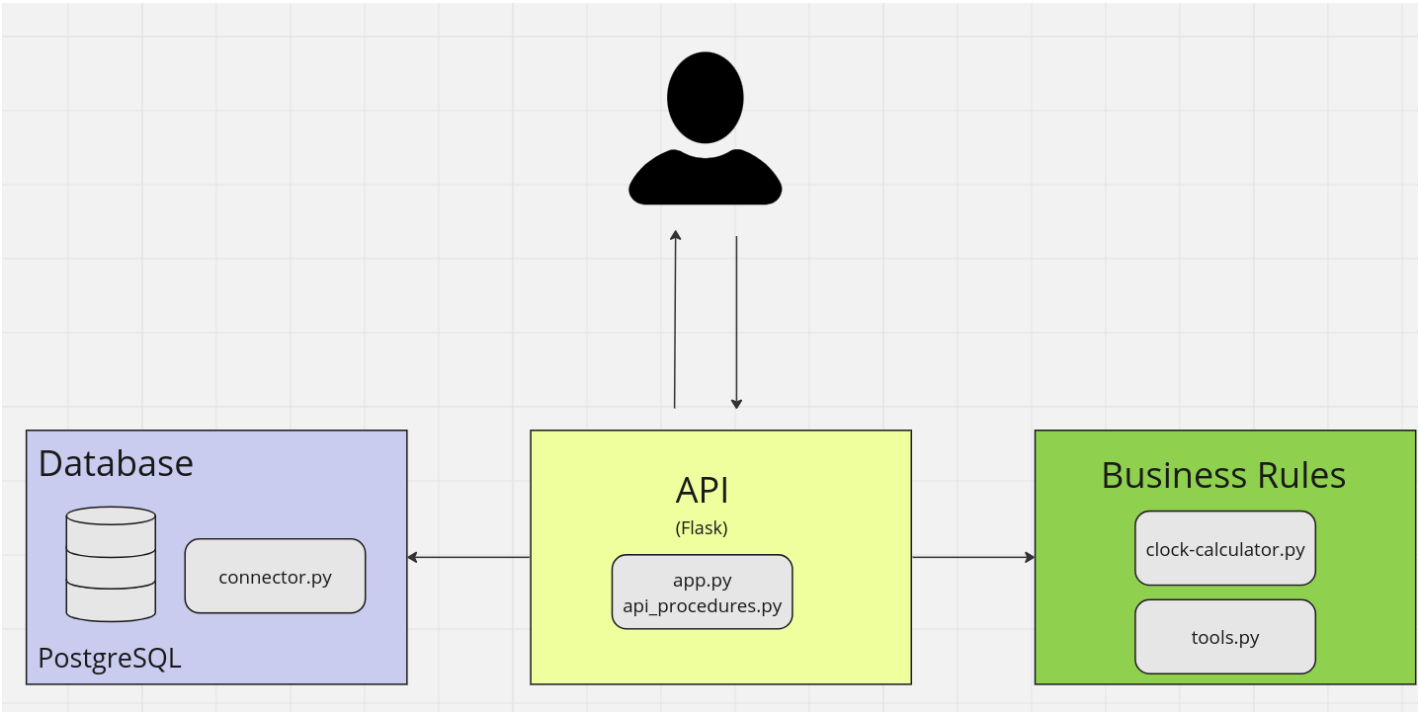
Informações Gerais:

O aplicativo “Clock API” foi implementado em Python e possui o objetivo de possibilitar interagir, via requisição http, com um sistema que determine o menor ângulo entre os dois ponteiros de um relógio em qualquer combinação de posições. A aplicação apresenta conexão com um banco de dados de forma a registrar todas as operações bem sucedidas e também testes automatizados funcionais para validar se há comportamento adequado.

Informações Técnicas

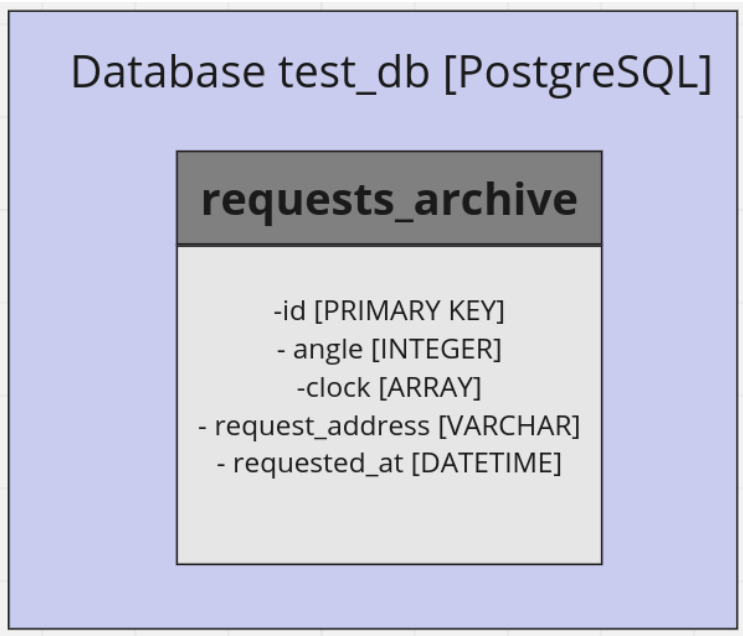
Estrutura:

Apesar de se tratar de um sistema de pequeno porte, a aplicação foi estruturada seguindo padrões desejáveis de arquitetura de software para favorecer sua escalabilidade e facilidade de manutenção. Assim, foi proposta a seguinte estrutura:



- **app.py:** Componente de execução do aplicativo. Responsável por reunir os diversos módulos e utilizá-los para a interação via API.
- **api_procedures.py:** Componente auxiliar do app.py, inicializando objetos e implementando funções necessárias para o funcionamento da api, em um módulo separado para isolamento de complexidade.
- **clock_calculator.py:** Contém as funções encarregadas de realizar o cálculo do menor ângulo entre os ponteiros do relógio, conforme as regras determinadas na tarefa.
- **utils.py:** Responsável por armazenar funções diversas necessárias para a execução das regras de negócio, mas que estão além do cálculo do menor ângulo.
- **connector.py:** Contém as funções que permitam interação com o banco de dados para persistência. Construído utilizando o banco PostgreSQL.

Foi tomado o cuidado de alcançar o máximo desacoplamento entre componentes para que fosse obtido um sistema modular. Assim, apesar de ter sido utilizado o banco de dados PostgreSQL, esse poderia ser substituído por qualquer variação, relacional ou não, apenas adaptando o arquivo "*connector.py*". O mesmo é válido para a API utilizada, elaborada a partir de Flask mas que poderia ser alterada para qualquer opção mais conveniente.



Devido à natureza simples do problema foi utilizada apenas uma tabela registrando o ângulo calculado como solicitado, entretanto com a adição de uma coluna contendo a disposição dos ponteiros do relógio utilizados para obter o resultado (*clock*), uma coluna para identificação do cliente da requisição (*request_address*) e uma para registro do horário em que a operação foi

feita. Além disso, optou-se por incluir uma coluna com o *id* como chave primária para que também possa servir de chave estrangeira em operações JOIN com outras tabelas que possam ser adicionadas, permitindo uma busca mais eficiente entre tabelas do banco de dados.

Para armazenamento no banco de dados é necessário que a base de dados *"test_db"* esteja previamente criado no sistema. Antes da inserção dos dados é verificado se a tabela apropriada existe, situação de interesse apenas no escopo dessa atividade e que não é recomendado em produção por criar com o banco uma interação desnecessária, entretanto o mesmo não é possível para o database. A solução proposta nesse caso seriam dois comandos em sequência, o primeiro para eliminar a base atual com aquele nome e o segundo para criá-la novamente, com efeito colateral de todos os dados armazenados serem perdidos a cada nova execução do programa, o que é contraditório com o objetivo de um banco de dados. Portanto, como essa não é uma situação esperada em um ambiente de produção, optou-se para esse teste configurar o banco anteriormente.

Funcionamento:

Ao ser inicializado em um servidor, o sistema passa a operar na porta determinada no arquivo .env contendo as variáveis de ambiente, à espera de uma requisição externa contendo dois parâmetros de entrada: valor da hora indicada e do minuto indicado pelos ponteiros do relógio, os dois como *strings*. Caso o minuto não seja determinado, será utilizado para ele o valor 0, mas o valor da hora é um input obrigatório.

Ao receber esses parâmetros, a primeira etapa é de validação dos inputs: uma vez que espera-se receber apenas os valores correspondentes aos indicados em um relógio de ponteiro, qualquer input que não pertença a esse conjunto deve gerar um erro com uma mensagem explicativa. Isso vale tanto para valores não numéricos quanto para numéricos que não se enquadrem nos critérios previstos, avaliados pela função *validate_parameters* e que retorna os inputs validados ou uma mensagem de aviso de erro, que será encaminhada ao usuário no formato JSON como resposta da requisição, se for o caso.

Caso sejam válidos os parâmetros de entrada é iniciada a etapa seguinte, que é a obtenção do menor ângulo entre os ponteiros, para a qual foram determinadas as seguintes regras de negócio:

- O ponteiro das horas se desloca 30° a cada hora e 0.5º a cada minuto, enquanto o dos minutos se desloca 6º por minuto, sendo o ângulo de cada determinado de forma independente.
- A posição de referência de cada ponteiro é o apontado para o 12, 0º, sendo os ângulos de cada ponteiro determinados com relação a ele. Dessa forma, com uma referência em comum o ângulo entre eles pode ser obtido por meio de uma subtração.
- Caso a diferença absoluta entre os dois ângulos seja menor que 180º (ex: 03:00), o menor ângulo será essa diferença. Caso seja maior que 180º (09:00), o menor ângulo será a

diferença entre 360º e a diferença dos ângulos obtida anteriormente. Esse raciocínio é intuitivo, uma vez que o "maior menor ângulo" possível é de 180º, considerado em sentido horário, sendo por isso o menor ângulo o valor em sentido anti-horário caso a diferença de posições seja maior que o maior possível.

Essas regras de negócio estão implementadas nas funções no módulo *clock_calculator.py*.

Após o cálculo de acordo com os critérios determinados, a operação é salva no banco PostgreSQL através de um método da instância *cursor*. Optou-se por criar uma classe própria para essa funcionalidade com o objetivo de criar uma camada extra que torne o sistema mais modular, sendo possível trocar o tipo de persistência para outro SQL flavour, ou mesmo para um banco de dados não estruturado se for conveniente, através unicamente de alterações na implementação da classe dB_Cursor, apenas alterando o módulo *connector.py*.

Por fim, caso todo o processo seja executado corretamente é retornada à requisição a resposta com o ângulo de interesse, convertida no formato JSON. Caso exista algum erro, a mensagem de erro é enviada como resposta ao usuário, também em JSON.

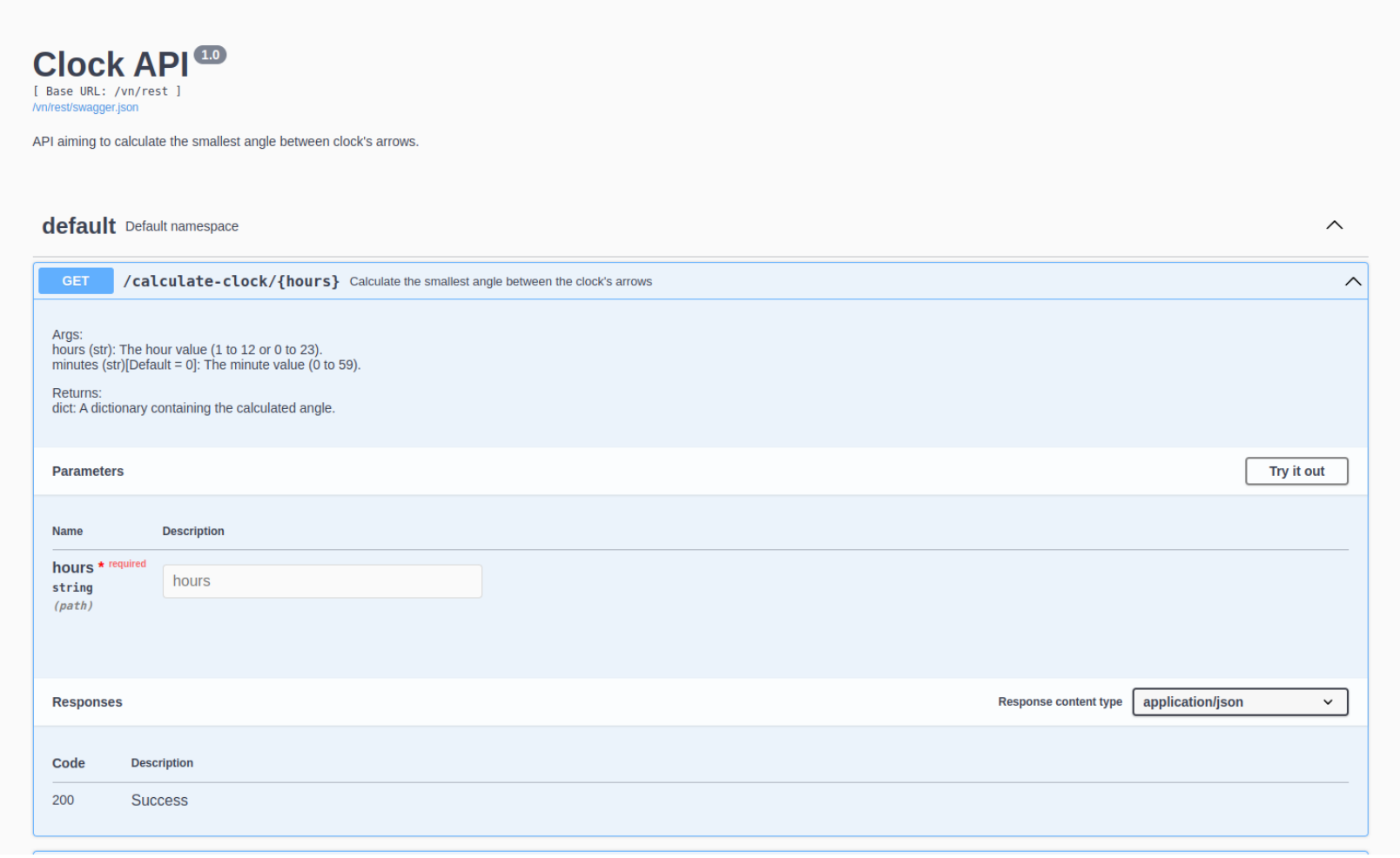
Como solicitado no enunciado do projeto, foi ainda implementada uma ferramenta de *caching* com memória de 600 segundos, ou seja, durante esse período os inputs da requisição e sua resposta são armazenados na memória RAM para que não precise ser calculada novamente caso a mesma requisição seja feita novamente. Essa é uma solução útil para aprimoramento do desempenho de uma API, uma vez que é comum que determinados recursos ocupem o interesse da maioria dos usuários de uma aplicação, evitando que capacidade de processamento seja utilizada diversas vezes para servir respostas iguais.

ChatGPT

Com o objetivo de tornar mais rápido o processo de formatação dos scripts, assim como a adição de docstrings e typehints (práticas recomendadas para facilitar a posterior manutenção de um sistema), foi utilizada a ferramenta ChatGPT 3.5.

Swagger

Está disponível aos desenvolvedores que necessitarem utilizar a aplicação a ferramenta "Swagger" embutida no Flask, que permite de forma interativa testar e esclarecer dúvidas a respeito do funcionamento do programa, como descrito na imagem, a seguir:



Para acessar o Swagger, basta enviar uma requisição no navegador para a URL raiz da API (*localhost:<porta>*)

Testes:

Foi solicitado no enunciado do teste a implementação de testes unitários, entretanto por essa ser uma parte importante de qualquer sistema foram implementadas avaliações não apenas das funcionalidades ligadas às regras de negócio(testes unitários) como também relacionando diferentes componentes do sistema (testes de integração), disponíveis no componente tests\unit_tests.py e tests\int_tests.py. Com o objetivo de tornar mais compacto os arquivos contendo os testes, adotou-se a ferramenta do *pytest* para elaboração de *fixtures* em uma módulo à parte, disponível em tests\conftest.py, conforme recomendado em sua documentação https://docs.pytest.org/en/7.1.x/reference/fixtures.html.

Por se tratar de um projeto de pequeno porte, não foi utilizada a metodologia TDD (*Test Driven Development)*, uma vez que a vantagem dessa técnica se encontra no ganho de tempo em médio e longo prazo ao denunciar o surgimento de erro devido à integração de sistemas complexos. Por ser um sistema simples, as interações são mais previsíveis e portanto foi mais eficiente a implementação tradicional, no entanto o TDD poderia ser utilizado por razões didáticas.

Dependências:

Para evitar possíveis problemas de gestão de dependências do código, foram utilizadas o mínimo de bibliotecas externas que possibilitasse a tarefa de ser cumprida, estando elas apresentadas a seguir:

Pacote	**Versão**
Flask	2.3.2
Flask_Caching	2.0.2
flask_restx	1.1.0
psycopg2_binary	2.9.6
pytest	7.4.0
python-dotenv	1.0.0

Sugestões de melhorias:

Devido aos requisitos da tarefa são persistidos apenas os dados gerados quando o funcionamento ocorre corretamente, havendo perda de toda a informação ao haver alguma falha

no processo. Esses registros podem ser valiosos se houver necessidade de otimizar o sistema, pois podem indicar os tipos de erros mais comuns e circunstâncias geradoras, pontos que podem direcionar os próximos passos de melhoria e manutenção para um sistema cada vez mais disponível. Portanto, para se manter o registro desses eventos recomenda-se algum tipo de banco de dado não estruturado como dynamoDB, ou mesmo um simples armazenamento via bucket no AWS S3, que já possibilitaria análise dessa informação através de ferramentas com esse fim, como AWS Athena.

Além de registrar a informação exibida no caso de erros, pode também ser desejável que o responsável pelo sistema seja informado prontamente em caso de falha, para a situação de um erro que exija intervenção imediata, como por exemplo servidor fora do ar. Para tal propõe-se a criação de um tópico específico utilizando o serviço AWS SNS ou equivalente, encaminhando e-mail ou mensagem SMS ao responsável.

O conceito de implementação de mensagens no SNS pode ser utilizado ainda para alertar situações de menor urgência, como implementar um alerta associado a um medidor da duração de execução de todo o processo. Dessa forma, caso o tempo médio para realizar a tarefa aumente, o responsável pela disponibilidade do sistema seria informado, uma vez que essa pode ser uma indicação de que a infraestrutura atual não está sendo adequada para gerenciar a carga de requisições solicitadas em determinados períodos, exigindo que seja providenciado melhoria na infraestrutura ou alteração no código para um processo mais eficiente.

Guia de instalação

Todos os comandos podem ser executados pelo prompt de comando da máquina utilizada. Para isso, basta inicializar o prompt e alterar o diretório para o local onde se deseja instalá-lo com o comando *"cd path\to\folder"*. Existem duas possíveis formas de utilização, que são:

1 - Execução local em ambiente virtual

	Tarefa	Comando
1	Verificar se as ferramentas estão instaladas	*$ git –version* *$ pip –version*
2	Clonar o repositório contendo o código fonte	*$ git clone https://github.com/aogdrummond/konv_mini_bank.git*
3	Criação do ambiente virtual	*$ python3 -m venv clock_api_venv*
4	Inicialização do ambiente virtual	*$ venv/Scripts/activate* (Windows) ou *$ venv/bin/activate* (Linux)
5	Instalação das dependências no ambiente virtual	*$ pip install -r requirements.txt*
6	Executar a aplicação	*$ python3 app.py*

2 - Execução em ambiente containerizado (atualmente sem persistência)

	Tarefa	Comando
1	Verificar se as ferramentas estão instaladas	*$ docker -v* *$ docker-compose -v* *$ git –version* *$ pip –version*
2	Clonar o repositório contendo o código fonte	*$ git clone https://github.com/aogdrummond/konv_mini_bank.git*

	Tarefa	Comando
3	Montagem da imagem com *Docker*	*$ docker build –tag clock-api .*
4	Execução do container	*$ docker run -d -p 5000:5000 clock-api*

3 - Testes automatizados

	Tarefa	Comando
1	Execução de testes automatizados	*$ pytest*