Implementação IOMT

André Gasoli Sichelero (136235), Carlos Eduardo Rosa Batista (193570), Dalton Oberdan Adiers (193851), Henrique Linck Poerschke (179791), Lucas Muliterno Tomasini Friedrich (168238) e Otávio Augusto Ficagna (195749).

Introdução

Tópicos

- Nomenclatura de projeto
- Divisão das funções
- Metodologia
- Diagramas
- Estrutura do projeto
- APIs
- Sensores e listener
- Interfaces web
- Mobile
- Gerenciamento da Cloud e Deploy
- Considerações finais

Nomenclatura de projeto

- **data_service:** Serviço que gerência o métodos (CRUD) voltados para as medições dos usuários.
- **user_service:** Serviço que gerência o métodos (CRUD) voltados para a gerência dos usuários.
- **web-data-interface:** Interface para o usuário fazer a gerência das suas medições.
- **interface-usuario:** Interface de gerência do usuário.
- **sensor_rest:** Sensor que gera as informações e envia via REST.
- **sensor_udp:** Sensor que gera as informações e envia via UDP.
- **listener_udp:** Middleware que recebe as informações do sensor UDP e faz a conversão para uma comunicação REST.

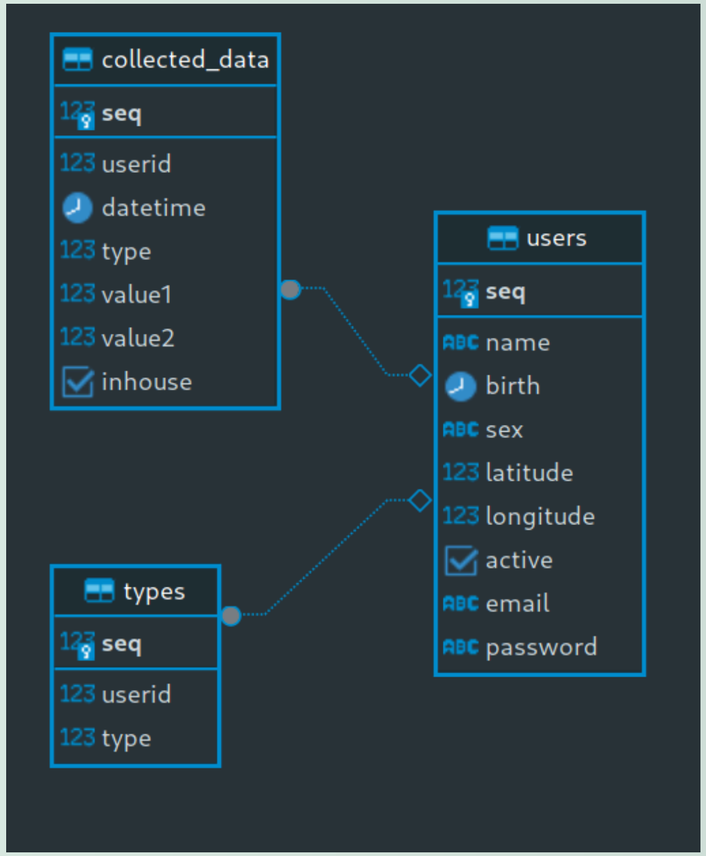
Divisão das funções

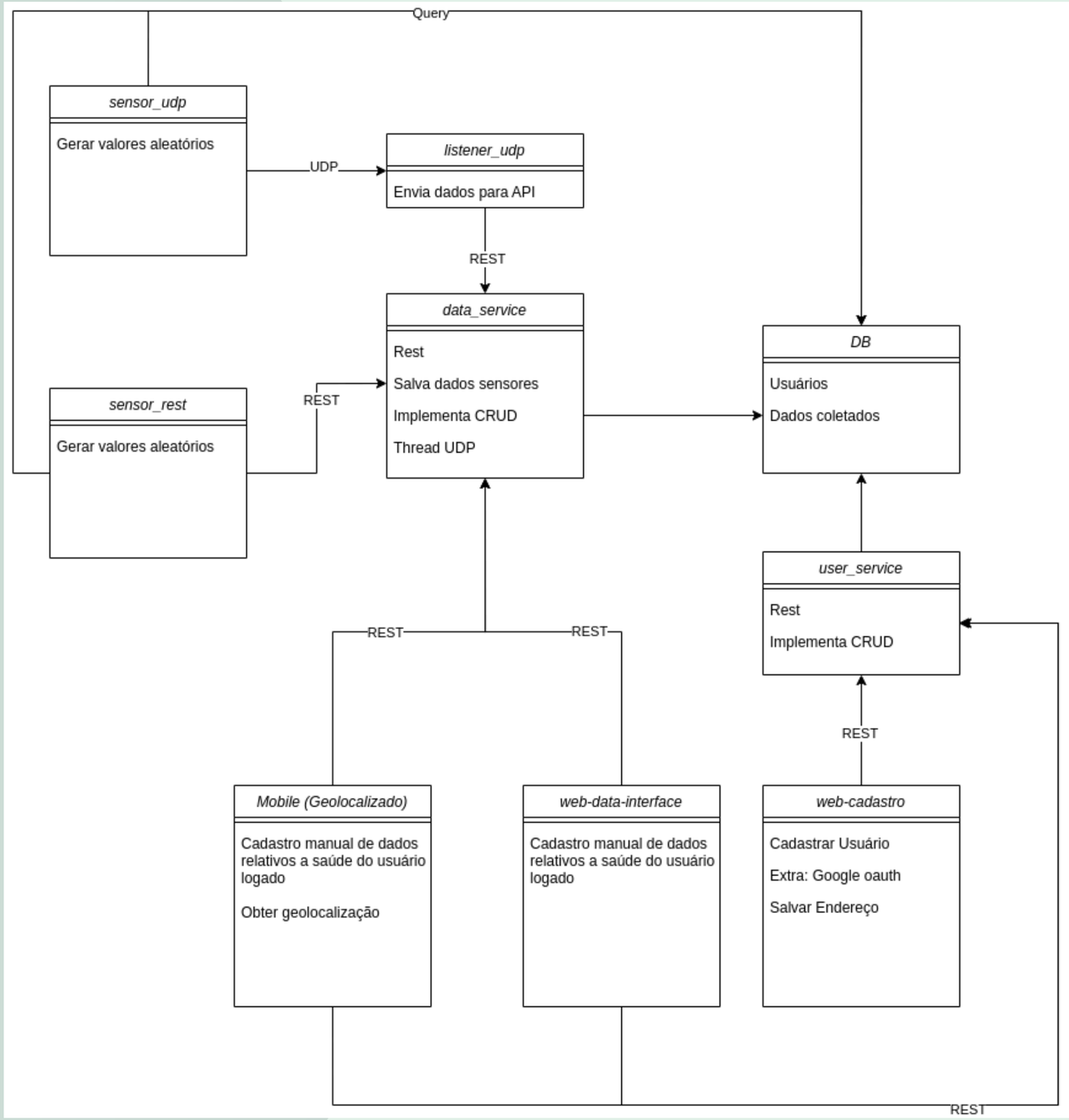
- **Otávio:** Líder do grupo, desenvolvimento do data_service, gerência da cloud e demais necessidades.
- **Lucas:** Desenvolvimento do data_service e user_service.
- **Henrique:** Desenvolvimento do user_service e interface-usuario.
- **Carlos:** Desenvolvimento da web-data-interface, interface-usuario e Mobile.
- **Dalton:** Desenvolvimento dos sensores e listener.
- **André:** Desenvolvimento dos sensores e listener.

Metodologia

- Durante o desenvolvimento do projeto obtamos por utilizar uma métodologia ágil voltada para o método SCRUM, porém com reuniões semanais no lugar de diárias por conta da disponibilidade dos integrantes.

Diagramas


collected_data
seq
userid
datetime
type
value1
value2
inhouse
users
seq
name
birth
sex
latitude
longitude
active
email
password
types
seq
userid
type


Query
sensor_udp
Gerar valores aleatórios
UDP
listener_udp
Envia dados para API
REST
data_service
Rest
Salva dados sensores
Implementa CRUD
Thread UDP
DB
Usuários
Dados coletados
sensor_rest
Gerar valores aleatórios
REST
REST
REST
user_service
Rest
Implementa CRUD
REST
Mobile (Geolocalizado)
Cadastro manual de dados relativos a saúde do usuário logado
Obter geolocalização
web-data-interface
Cadastro manual de dados relativos a saúde do usuário logado
web-cadastro
Cadastrar Usuário
Extra: Google oauth
Salvar Endereço
REST

Estrutura do projeto

Estrutura dos diretórios:

- Conforme requisitos o projeto foi utilizado o GitHub para organização do projeto.
- Na estrutura de diretórios temos as seguintes pastas raiz:
  - **api:** Guarda os fontes e demais arquivos dos serviços de usuário e medições.
  - **db:** Guarda arquivos referentes a o banco de dados.
  - **mobile:** Guarda o APK do PWA mobile.
  - **sensores:** Guarda os fontes e demais arquivos dos sensores e listener.
  - **web:** Guarda os arquivos referentes as interfaces web e mobile.

```
simuladorIoMT
├── api
│   ├── data_service
│   └── user_service
├── db
│   └── Tables.sql
├── mobile
│   └── IOMT - Interface Location.apk
├── sensores
│   ├── docker-compose.yml
│   ├── requirements.txt
│   ├── sensor_rest
│   ├── sensor_udp
│   └── udp_listener
└── web
    ├── interface-usuario
    └── web-data-interface
```

Estrutura do projeto

Segurança:

- **Geral do projeto:** Todo o projeto utiliza arquivos no padrão ".env" para guardar as "environments". Dessa forma nenhuma váriavel como senhas, informações sobre token e dados de conexão ao banco de dados será encontrada diretamente no código ou no nosso repósitório do GitHub, mantendo assim os padrões de segurança.
- **Das APIs:** As APIs implementam login por token e hash de senha. Onde via login conseguem diferenciar usuário comuns do administrador e mostrar apenas os dados corretos do usuário logado. Em nuvem utilizam protocolo HTTPS para comunicação.
- **Das interfaces web:** As interfaces web implementam chamadas via token para as APIs especificando assim o usuário logado. Estão hospedadas com protocolo HTTPS para comunicação.
- **Do banco de dados:** O banco de dados só é acessível internamente pela cloud ou utilizando proxy com chaves ssl.

Estrutura do projeto

Segurança:

- **Da cloud:** Para comunicação interna entre os itens hospedados na cloud foi utilizada a estrutura do Virtual Private Cloud (VPC) da GoogleCloud. Isolando principalmente a comunicação entre as VMs e o banco de dados em uma rede privada. Já no caso das APIs no CloudRun foi utilizado as ferramentas da própria ferramenta que permite efetuar um link entre a API e instância do CloudSQL utilizando environments para conexão.

Estrutura do projeto

Considerações:

- Dentro de todos os componentes do projeto podemos encontrar arquivos readme.md que descrevem como o projeto pode ser rodado localmente e demais documentações. Dessa forma a apresentação atual não vai citar esse tipo de informação.

APIs

Principais tecnologias e bibliotecas usadas:

- **Python:** Linguagem utilizada para construção da API.
- **FastApi:** Biblioteca utilizada para implementação de toda a lógica REST da API.
- **SQLAlchemy:** Biblioteca utilizada para gerência da persistência e conexão ao banco de dados.
- **Dotenv:** Biblioteca utilizada para obtenção das "envs" em ambiente de dev.
- **Pydantic:** Biblioteca utilizada para gerenciar a "tipagem" no Python.
- **Datetime/Pytz:** Bibliotecas para gerenciamento de data e hora.
- **JWT:** Biblioteca utilizada para criação do token de acesso.
- **Passlib:** Biblioteca utilizada para hash e verificação das senhas dos usuários.

APIs

Explicação da estrutura:

- Para manter o padrão do projeto ambas as apis (data_service e user_service) utilizam as mesmas estruturas e nomenclaturas:

user_service

```
└── app
    ├── config.py
    ├── database.py
    ├── Dockerfile
    ├── models.py
    ├── requirements.txt
    ├── security.py
    ├── user_actions.py
    └── user_service.py
```

data_service

```
└── app
    ├── collected_data_actions.py
    ├── collected_data_service.py
    ├── config.py
    ├── database.py
    ├── Dockerfile
    ├── models.py
    ├── requirements.txt
    └── security.py
```

APIs

Explicação da estrutura:

- **service:** Contém os enpoints de GET, POST, PUT e DELETE da API. Onde os métodos dependem do usuário logado (TOKEN) e da conexão ao banco de dados. (Obs: Acesse /docs das APIs para verificar todos os endpoints)
- **actions:** Contém a lógica de todos os métodos do service, sendo responável por formatar as saídas da API, tratar erros, utilizar a persitência com os métodos do SQLAlchemy e verificar as permissões do usuário logado (Admin ou não).
- **config:** Utilizado para carregar e gerenciar as envs.
- **security:** Utilizado para guardar a lógica de criação/gerencia do token, obtenção do usuário logado e verificação/hash de senha.
- **database:** Gerencia as conexões ao banco de dados durante as requisições.
- **models:** Contém as classes utilizadas para persistência e inputs.
- **Dockerfile:** Arquivo Docker para utilização da API em container.
- **requirements:** Contém todos os requisitos necessários que devem ser instalados para utilização da API.

APIs

Features:

- As apis implementam todas as features constantes na especificação do trabalho.
- Como features adicionais podemos citar:
  - Persistência de dados.
  - Token.
  - Verificação das permissões do usuário (Admin ou não).
  - Na api de cadastro utilização da API do OpenStreetMaps para durante o cadastro buscar automaticamente a latidute e longitude do usuário pelo seu endereço.
  - Gerência das sessões de conexão com o banco utilizando a propriedade yield da FastAPI o que garante o fechamento automático das sessões após cada requisição.

Sensores

Principais tecnologias e bibliotecas usadas:

- **Python:** Linguagem utilizada para construção dos sensores.
- **Psycopg2:** Utilizado para conexão com o banco de dados.
- **Dotenv:** Biblioteca utilizada para obtenção das "envs" em ambiente de dev.
- **Requests:** Biblioteca de requests para as requisições rest.
- **Socket:** Biblioteca para uso de sockets no Python.
- **Pickle:** Biblioteca de serialização e deserialização de objetos no Python.

Sensores

Explicação da estrutura:

- Para manter o padrão do projeto e facilitar o deploy os sensores e listener utilizam pastas próprias contendo seus arquivos necesseários:

```
sensores
├── docker-compose.yml
├── requirements.txt
├── sensor_rest
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── sensor_rest.py
│   └── util.py
├── sensor_udp
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── sensor_udp.py
│   └── util.py
├── udp_listener
│   ├── Dockerfile
│   ├── requirements.txt
│   ├── udp_listener.py
│   └── util.py
```

Sensores

Explicação da estrutura:

- **util:** Presente em todos os sensores e listener, contém as principais funções necessárias para geração das medições e comunicação com o banco.
- **sensor_rest:** Contém a lógica do sensor rest, que a cada uma hora gera e envia as medições para a API data_service.
- **sensor_udp:** Contém a lógica do sensor udp, que a cada uma hora gera e envia as medições para o listener.
- **udp_listener:** Contém a lógica do listener, que fica ininterruptamente escutando conexões UDP e ao receber informação envia para a API data_service.
- **Dockerfile:** Arquivo Docker para utilização da API em container.
- **requirements:** Contém todos os requisitos necessários que devem ser instalados para utilização da API.
- **docker-compose:** Arquivo do Docker Compose que é utilizado para gerar e rodar os containers dos sensores e do listener criando uma bridge de conexão entre eles, permitindo comunicação UDP entre os mesmos.

Sensores

Features:

- Os sensores implementam todas as features constantes na especificação do trabalho.
- Como features adicionais podemos citar:
  - Geração automática das medições a cada uma hora.
  - Normalização dos dados.
  - Verificação dos usuários ativos no banco e quais tipos de informação eles estão inscritos para realizar a geração das medições.

Interfaces Web

Principais tecnologias e bibliotecas usadas:

- **Javascript:** Linguagem utilizada para construção das interfaces.
- **React:** Biblioteca para criação de interfaces utilizada nas interfaces.
- **Boostrap:** Framework frontend utilizada nas interfaces para facilitar a criação das mesmas.
- **Vite:** Ferramenta de build moderna para projetos front-end. Utilizado para facilitar e melhorar o build das interfaces.

Interfaces Web

Explicação da estrutura:

- A estrutura das interfaces web se baseia em duas pastas que dentro contém toda lógica de uma aplicação utilizando React com o Vite.

web
├── interface-usuario
└── web-data-interface

Interfaces Web

Features:

- As interfaces web implementam todas as features constantes na especificação do trabalho.
- Como features adicionais podemos citar:
  - Gerenciar as requisições as APIs mantendo o token do login guardado até sua expiração ou fim do logout do usuário.
  - Na interface de cadastro busca de endereço do usuário via o CEP do mesmo através da API dos Correios.
  - Na interface de medições busca a geolocalização do usuário (mesmo via navegador) para saber se o mesmo está em casa.

Mobile

Principais tecnologias e bibliotecas usadas:

- **PWA (Progressive Web App):** É uma abordagem para criar aplicativos móveis utilizando tecnologias web como HTML, CSS e JavaScript. Ele combina as vantagens de sites e apps nativos, oferecendo uma experiência similar à de um app, com funcionalidades como notificações push e acesso offline.
- Com a abordagem do PWA reutilizamos toda a nossa web-data-interface com as devidas configurações para efetuar a geração do nosso app mobile gerando um APK.

Mobile

Features:

- As apis implementam todas as features constantes na especificação do trabalho. E também as demais features das web interfaces por conta do PWA.

Cloud

Ferramentas utilizadas:

- Como provedor de cloud foi utilizado o GoogleCloud sendo que os serviços utilizados foram:
  - CloudRun: Plataforma gerenciada que permite executar aplicações containerizadas de maneira simples e escalável.
  - CloudSQL com PostgreSQL: Serviço de banco de dados em nuvem simples e escalável.
  - Instâncias de VM com ComputeEngine: Permite a criação de máquinas virtuais altamente configuráveis e escaláveis.
  - VPC

Cloud

Deploy:

- **Banco de dados:** Utilizamos o CloudSQL com o PostgreSQL.
- **data_service e user_service:** Para o deploy foi gerado containers e utilizado uma instância do CloudRun para cada um dos containers, além de utilizar a própria ferramenta do CloudRun para comunicação com o CloudSQL.
- **Interfaces web:** Para o deploy das interfaces web foi utilizado duas VMs do ComputeEngine, uma para cada interface. Após rodar o build das mesmas, utilizamos o servidor web Njinx instalado nas VMs para realizar o deploy. Para obtenção do certificado https utilizamos o serviço do No-Ip e o lets-encrypt.
- **Sensores:** Para o deploy dos sensores foi utilizado uma única VM do ComputeEngine combinada com o DockerCompose para permitir a comunição UDP entre os containers mesmo na cloud.

Extras Realizados

1 - Token nas APIs e interface.
2 - Latitude e longitude por endereço (Através das APIs dos Correios e OpenStreetMaps).
3 - Hospedagem do serviço em instâncias de servidores “nas nuvens”, acessando o BD na mesma nuvem.
4 - Hospedagem da aplicação, caso web, em instâncias de servidores “nas nuvens”.

Considerações Finais

Muito obrigado pela atenção!