Documentação de Manutenção - Aplicação de Monitoramento de Manobras de Navios

1. Introdução

Este documento tem como objetivo fornecer uma visão abrangente e detalhada da aplicação Python desenvolvida para monitorar manobras de navios no Porto do Rio de Janeiro. Ele é destinado a desenvolvedores que necessitam realizar manutenção, depuração ou futuras extensões do sistema. A aplicação é construída utilizando o framework Flask e realiza o scraping de dados de um website específico para apresentar informações atualizadas sobre o status da barra e as manobras de navios.

O principal propósito desta documentação é garantir que qualquer desenvolvedor com conhecimento em Python e Flask possa entender rapidamente a estrutura do código, suas funcionalidades principais, as dependências necessárias e como o sistema interage com fontes externas de dados. Serão abordados aspectos como a arquitetura do projeto, as funções de scraping, a lógica de detecção de conflitos e a exposição de dados via API e interface web.

Ao final deste documento, o leitor deverá ser capaz de:

- Compreender a finalidade e o escopo da aplicação.
- Identificar os principais módulos e funções do código.
- Entender o fluxo de dados desde o scraping até a apresentação.
- Realizar a configuração e execução do ambiente de desenvolvimento.
- Efetuar manutenções e solucionar problemas de forma eficiente.
- Propor e implementar novas funcionalidades com base na arquitetura existente.

Esta documentação será atualizada conforme novas funcionalidades forem adicionadas ou alterações significativas forem implementadas no código-fonte.

Recomenda-se que os desenvolvedores consultem a versão mais recente deste documento antes de iniciar qualquer trabalho de manutenção ou desenvolvimento.

2. Estrutura do Projeto e Dependências

2.1. Estrutura de Arquivos

O projeto consiste em um único arquivo Python principal (`app.py` ou similar, dependendo de como foi nomeado pelo usuário) que contém toda a lógica da aplicação Flask, incluindo as rotas, funções de scraping e processamento de dados. Além disso, espera-se a existência de um diretório `templates` para os arquivos HTML utilizados pelo `render_template` do Flask.

```
.
├── app.py (ou nome similar)
└── templates/
    └── index.html
```

2.2. Dependências

Para o correto funcionamento da aplicação, as seguintes bibliotecas Python são necessárias. Elas podem ser instaladas utilizando o `pip`, o gerenciador de pacotes do Python, a partir de um arquivo `requirements.txt` ou individualmente.

```
Flask
requests
bs4 (BeautifulSoup4)
lxml (parser para BeautifulSoup)
pytz
```

Exemplo de `requirements.txt`:

```
Flask==X.Y.Z
requests==X.Y.Z
beautifulsoup4==X.Y.Z
lxml==X.Y.Z
pytz==X.Y.Z
```

Instalação:

```
pip install -r requirements.txt
```

Ou individualmente:

```
pip install Flask requests beautifulsoup4 lxml pytz
```

3. Configuração e Execução

3.1. Variáveis de Ambiente

A aplicação utiliza a variável de ambiente `PORT` para definir a porta em que o servidor Flask será executado. Se não for definida, o valor padrão será `5000`.

Exemplo (Linux/macOS):

```
export PORT=8080
```

Exemplo (Windows - CMD):

```
set PORT=8080
```

Exemplo (Windows - PowerShell):

```
$env:PORT=8080
```

3.2. Execução da Aplicação

Para iniciar a aplicação, navegue até o diretório raiz do projeto e execute o arquivo Python principal:

```
python app.py
```

Se o `debug=True` estiver ativado no `if __name__ == "__main__":` (como está no código fornecido), o servidor será reiniciado automaticamente a cada alteração no código, o que é útil para o desenvolvimento.

Após a execução, a aplicação estará acessível em `http://localhost:5000` (ou na porta definida pela variável `PORT`).

4. Arquitetura do Código e Funções Principais

A aplicação segue uma arquitetura MVC (Model-View-Controller) simplificada, onde as funções Python atuam como o "Controller" e o "Model" (realizando o scraping e processamento de dados), e os templates HTML como a "View".

4.1. Variáveis Globais e Constantes

- `port`: Define a porta do servidor Flask, obtida da variável de ambiente `PORT` ou padrão `5000`.
- `app`: Instância da aplicação Flask.
- `URL`: URL base para o scraping (`https://www.praticagem-rj.com.br/`).
- `BERCOS_INCLUIR_TODOS`: Conjunto de berços de interesse para filtragem de navios.

4.2. Funções de Scraping e Processamento de Dados

`get_status_barra()`

- **Propósito:** Obtém o status atual da barra da Baía de Guanabara (restrita ou não) a partir do website.
- **Retorno:** Um dicionário com `restrita` (booleano) e `mensagem` (string).
- **Detalhes:** Realiza uma requisição GET para a `URL` principal, parseia o HTML com BeautifulSoup, busca por um `<span>` contendo "BAIA DE GUANABARA" e, a partir dele, encontra o `<div>` com o status da barra. Verifica se a mensagem contém "BARRA RESTRITA".

`get_all_navios_manobras()`

- **Propósito:** Realiza o scraping de todas as manobras de navios listadas na página e as processa.

- **Retorno:** Uma lista de dicionários, onde cada dicionário representa uma manobra de navio com diversas informações (data, hora, navio, calado, manobra, berço, IMO, tipo de navio, ícone, alerta, terminal).
- **Detalhes:**
  - Faz uma requisição GET para a `URL`.
  - Localiza a tabela principal de manobras (`tbManobrasArea`).
  - Itera sobre as linhas da tabela, extraindo dados de cada coluna.
  - **Filtragem:** Apenas navios em `BERCOS_INCLUIR_TODOS` são processados.
  - **Normalização de Dados:** Ajusta o formato da hora se necessário (ex: `HH:M` para `HH:MM`).
  - **Cálculo de Status e Alerta:** Calcula o status da manobra (`futuro`, `hoje`, `passado`) e gera alertas (`entrada_antecipada`, `entrada_futura`, `saida_futura`, `saida_atrasada`) com base na data/hora da manobra e na hora atual.
  - **Determinação de Ícone:** Atribui um ícone de acordo com o `tipo_navio`.
  - **Determinação de Terminal:** Classifica o navio em um terminal (`rio`, `multi`, `manguinhos`, `pg1`) com base no berço.
  - **Tratamento de Erros:** Inclui um bloco `try-except` para lidar com erros durante o processamento de cada linha.

`get_navios(terminal_filter='todos')`

- **Propósito:** Retorna uma lista de navios manobras, opcionalmente filtrada por terminal e removendo duplicatas.
- **Parâmetros:** `terminal_filter` (string): `todos`, `rio`, `multi`, `manguinhos`, `pg1`.
- **Retorno:** Uma lista de dicionários de manobras de navios, sem duplicatas.
- **Detalhes:** Chama `get_all_navios_manobras()`, aplica o filtro de terminal e, em seguida, remove entradas duplicadas usando um `set` de chaves (`data`, `hora`, `navio`, `manobra`).

`detectar_conflitos(navios_rio_manobras, navios_multi_manobras)`

- **Propósito:** Identifica possíveis conflitos de manobra entre navios dos terminais "rio" e "multi".
- **Parâmetros:**
  - `navios_rio_manobras` (lista de dicts): Manobras de navios do terminal "rio".
  - `navios_multi_manobras` (lista de dicts): Manobras de navios do terminal "multi".
- **Retorno:** Uma lista de dicionários, onde cada dicionário descreve um conflito encontrado.
- **Detalhes:**
  - Agrupa as manobras do terminal "rio" por nome de navio.
  - Para cada navio do terminal "rio", determina um período de ocupação (início da primeira entrada até o fim da última saída).
  - Compara este período com as janelas de manobra (entrada/saída) dos navios do terminal "multi".
  - Se houver sobreposição de períodos, um conflito é registrado, incluindo detalhes dos navios e manobras envolvidas.

4.3. Rotas da Aplicação Flask

`@app.route("/")` - `home()`

- **Método:** GET
- **Propósito:** Renderiza a página HTML principal (`index.html`) com os dados das manobras de navios e o status da barra.
- **Fluxo:**
  1. Obtém a hora atual e formata a `ultima_atualizacao` .
  2. Chama `get_all_navios_manobras()` para obter todos os dados.
  3. Filtra os navios para os terminais "rio" e "multi".
  4. Chama `detectar_conflitos()` para encontrar conflitos.

5. Atualiza os dicionários de navios do terminal "rio" com uma flag `conflito_porterne` e `conflito_manobra_tipo` se houver conflito.
6. Prepara uma lista `navios_para_exibir` removendo duplicatas para a exibição.
7. Chama `get_status_barra()`.
8. Renderiza `index.html` passando os dados de `navios`, `ultima_atualizacao`, `barra_info` e `terminal_selecionado`.

`@app.route("/api/navios")` - `api_navios()`

- **Método:** GET
- **Propósito:** Fornece os dados das manobras de navios e o status da barra em formato JSON, com opção de filtragem por terminal.
- **Parâmetros (Query String):** `terminal` (string, opcional): `todos`, `rio`, `multi`, `manguinhos`, `pg1`. Padrão é `todos`.
- **Fluxo:**
  1. Similar à rota `home()`, obtém a hora atual e `ultima_atualizacao`.
  2. Obtém o filtro de terminal da requisição.
  3. Chama `get_all_navios_manobras()`.
  4. Filtra os navios para os terminais "rio" e "multi" e detecta conflitos, marcando-os nos dados.
  5. Prepara `navios_para_exibir` aplicando o filtro de terminal e removendo duplicatas.
  6. Chama `get_status_barra()`.
  7. Retorna um JSON contendo `navios`, `ultima_atualizacao`, `barra_info` e `conflitos`.

4.4. Bloco de Execução Principal

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=port, debug=True)
```

Este bloco garante que a aplicação Flask seja executada apenas quando o script é invocado diretamente (não quando importado como módulo). Ele configura o servidor

para escutar em todas as interfaces de rede (`0.0.0.0`) na porta definida pela variável `port` e ativa o modo de depuração (`debug=True`), que é útil para o desenvolvimento, mas **deve ser desativado em produção** por questões de segurança e performance.

5. Considerações para Manutenção

Esta seção aborda pontos críticos e recomendações para a manutenção eficaz da aplicação.

5.1. Robustez do Scraping

O scraping de dados é a parte mais sensível da aplicação, pois depende diretamente da estrutura HTML do website `https://www.praticagem-rj.com.br/`. Qualquer alteração na estrutura das tabelas, IDs de elementos, classes CSS ou textos utilizados para identificação pode quebrar o scraping.

Pontos de atenção:

- **Alterações no HTML:** Monitore o website de origem para identificar mudanças na estrutura. Se o scraping parar de funcionar, o primeiro passo é inspecionar o HTML da página para verificar se os seletores (`id` , `class` , `string` em `re.compile`) ainda são válidos.
- **Mensagens de Erro:** As funções de scraping (`get_status_barra` , `get_all_navios_manobras`) incluem blocos `try-except` para capturar erros. Verifique os logs da aplicação para mensagens de erro relacionadas ao scraping.
- **User-Agent:** Em alguns casos, websites podem bloquear requisições sem um `User-Agent` válido. Se o `requests.get()` começar a falhar, considere adicionar um cabeçalho `User-Agent` à requisição:

  `python headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'} response = requests.get(URL, headers=headers)`

5.2. Lógica de Negócio e Regras de Conflito

A lógica de detecção de conflitos na função `detectar_conflitos` é complexa e baseada em regras específicas de tempo e tipo de manobra. Qualquer alteração nessas regras ou a necessidade de incluir novos tipos de conflito exigirá uma revisão cuidadosa desta função.

Pontos de atenção:

- **Períodos de Ocupação:** A forma como os `periodo_inicio_rio` e `periodo_fim_rio` são calculados é crucial. Certifique-se de que a lógica de `timedelta` e a ordenação das manobras (`manobras_rio.sort`) estejam corretas para o cenário desejado.
- **Janelas de Conflito:** A `janela_multi_inicio` e `janela_multi_fim` (atualmente +/- 1 hora da manobra) definem a sensibilidade da detecção de conflitos. Ajustes podem ser necessários.
- **Tipos de Manobra:** A função considera apenas manobras de `ENTRADA` (`E`) e `SAÍDA` (`S`) para a detecção de conflitos. Se outros tipos de manobra (`M` - Mudança) precisarem ser considerados, a lógica deve ser expandida.

5.3. Gerenciamento de Tempo e Fuso Horário

A aplicação lida com datas e horas, incluindo a conversão para o fuso horário de São Paulo (`America/Sao_Paulo`). É fundamental garantir que todas as operações com datas e horas estejam corretas para evitar erros de cálculo de status ou alertas.

Pontos de atenção:

- **`pytz`:** A biblioteca `pytz` é utilizada para lidar com fusos horários. Certifique-se de que o fuso horário configurado (`America/Sao_Paulo`) é o correto para o contexto da aplicação.
- **`datetime.now()` vs. `datetime.now(tz)`:** Sempre que a hora atual for relevante para comparações com manobras, utilize `datetime.now(tz)` para garantir que ambas as datas/horas estejam no mesmo fuso horário ou sejam *timezone-aware*.
- **Formato de Data/Hora:** A extração e formatação de `data_hora` do HTML e a subsequente conversão para objetos `datetime` (`navio_date`) são pontos críticos. Erros de parsing podem levar a datas incorretas.

5.4. Performance e Escalabilidade

Para uma aplicação de scraping, a performance pode ser um gargalo, especialmente se o volume de dados ou a frequência de requisições aumentar.

Recomendações:

- **Cache:** Considere implementar um mecanismo de cache para os resultados do scraping. Isso evitaria requisições repetidas ao website de origem, reduzindo a carga no servidor remoto e acelerando as respostas da sua aplicação. Bibliotecas como `requests-cache` podem ser úteis.
- **Assincronicidade:** Para scraping de múltiplos recursos ou para melhorar a responsividade da API, a utilização de bibliotecas assíncronas (ex: `httpx` com `asyncio`) pode ser considerada, embora exija uma reestruturação significativa do código.
- **Monitoramento:** Monitore o tempo de resposta das funções de scraping e das rotas da API para identificar gargalos de performance.

5.5. Segurança

- **Modo Debug:** O `debug=True` no `app.run()` **NÃO DEVE SER USADO EM PRODUÇÃO**. Ele expõe informações sensíveis e permite a execução de código arbitrário. Para produção, remova `debug=True` e utilize um servidor WSGI robusto como Gunicorn ou uWSGI.
- **Validação de Entrada:** Embora esta aplicação não receba muitas entradas do usuário diretamente, em sistemas mais complexos, a validação de todas as entradas é crucial para prevenir ataques como injeção de código ou XSS.

5.6. Testes

Para garantir a estabilidade e a correção da aplicação, especialmente após alterações, a implementação de testes automatizados é altamente recomendada.

- **Testes Unitários:** Teste funções individuais como `get_status_barra`, `get_all_navios_manobras` e `detectar_conflitos` com dados mockados (simulados) para garantir que a lógica interna funcione como esperado.

- **Testes de Integração:** Teste as rotas da API (`/` e `/api/navios`) para garantir que a integração entre as funções e o Flask esteja correta e que as respostas estejam no formato esperado.

5.7. Logs

Utilize um sistema de logging mais robusto do que simples `print()` para registrar eventos, erros e informações de depuração. Isso facilita a identificação de problemas em ambientes de produção.

```
import logging

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %
(message)s')

# Exemplo de uso:
# logging.info("Aplicação iniciada")
# logging.error(f"Erro ao processar linha: {e}")
```

6. Conclusão

Esta documentação buscou cobrir os aspectos mais relevantes da aplicação de monitoramento de manobras de navios, desde sua estrutura básica até considerações avançadas para manutenção e otimização. A compreensão aprofundada das funções de scraping, da lógica de detecção de conflitos e das rotas da API é fundamental para qualquer intervenção no código.

É crucial que os desenvolvedores encarregados da manutenção estejam cientes da dependência da aplicação em relação à estrutura do website de origem e que implementem testes robustos para garantir a estabilidade e a correção das funcionalidades. A adoção de boas práticas de desenvolvimento, como o uso de logging adequado e a desativação do modo de depuração em produção, contribuirá significativamente para a longevidade e a confiabilidade do sistema.

Em caso de dúvidas ou necessidade de aprofundamento em tópicos específicos, recomenda-se consultar a documentação oficial das bibliotecas utilizadas (Flask, requests, BeautifulSoup, pytz) e, se possível, entrar em contato com o desenvolvedor original ou a equipe responsável pelo projeto.

7. Referências

[1] Documentação Oficial do Flask: https://flask.palletsprojects.com/ [2] Documentação Oficial do Requests: https://docs.python-requests.org/en/master/ [3] Documentação Oficial do BeautifulSoup: https://www.crummy.com/software/BeautifulSoup/bs4/doc/ [4] Documentação Oficial do pytz: https://pythonhosted.org/pytz/ [5] Website de Origem dos Dados: https://www.praticagem-rj.com.br/