Guia de Arquitetura do Phishing Manager

1. Visão Geral da Arquitetura

O Phishing Manager é uma aplicação web robusta e escalável, projetada para gerenciar campanhas de phishing de forma eficiente e segura. A arquitetura do sistema é dividida em duas partes principais: um **Backend** construído com Flask (Python) e um **Frontend** desenvolvido em React (JavaScript). Essa separação permite o desenvolvimento independente, maior flexibilidade na escolha de tecnologias e melhor escalabilidade.

1.1. Princípios de Design

- **Separação de Preocupações (SoC)**: O backend é responsável pela lógica de negócios, persistência de dados e segurança da API, enquanto o frontend lida com a interface do usuário e a experiência do usuário.
- **API-First**: A comunicação entre o frontend e o backend é realizada exclusivamente através de uma API RESTful bem definida, garantindo interoperabilidade e facilitando a integração com outros sistemas.
- **Modularidade**: O código é organizado em módulos lógicos e reutilizáveis, promovendo a manutenibilidade e a extensibilidade.
- **Segurança em Primeiro Lugar**: Medidas de segurança robustas são implementadas em todas as camadas da aplicação, desde a validação de entrada até a autenticação de dois fatores e proteção contra ataques comuns.
- **Testabilidade**: A arquitetura é projetada para facilitar a escrita de testes unitários, de integração e end-to-end, garantindo a qualidade e a estabilidade do software.
- **Observabilidade**: O sistema inclui mecanismos de logging detalhados e monitoramento para facilitar a depuração e a operação em produção.

1.2. Componentes Principais

1.2.1. Backend (Flask)

O backend é o coração da aplicação, responsável por: - Gerenciamento de usuários e autenticação (incluindo 2FA). - Lógica de negócios para campanhas de phishing, modelos, domínios e URLs geradas. - Interação com o banco de dados. - Exposição de uma API RESTful para o frontend e outros clientes. - Serviços de segurança (rate limiting, sanitização de entrada, logging de segurança). - Integração com serviços externos (e.g., Telegram para notificações).

1.2.2. Frontend (React)

O frontend é a interface do usuário, responsável por: - Apresentação dos dados e interação com o usuário. - Consumo da API RESTful do backend. - Gerenciamento de estado da aplicação. - Experiência do usuário responsiva e intuitiva.

1.2.3. Banco de Dados

O sistema utiliza um banco de dados relacional para persistir todos os dados da aplicação, como usuários, campanhas, modelos, logs, etc. A escolha padrão é SQLite para facilidade de implantação, mas a arquitetura permite a fácil migração para outros bancos de dados como PostgreSQL ou MySQL.

1.2.4. Instalador Aprimorado

Um instalador robusto e interativo que facilita a implantação do Phishing Manager em diferentes ambientes, incluindo instalação manual, via Docker ou como serviço SystemD.

2. Detalhamento do Backend (Flask)

O backend Flask segue uma estrutura modular, com componentes bem definidos para cada funcionalidade.

2.1. Estrutura de Diretórios

```
phishing-manager/
├── src/
│   ├── __init__.py
│   ├── main.py             # Ponto de entrada da aplicação Flask
│   ├── config.py           # Configurações da aplicação
│   ├── models/             # Definições dos modelos de banco de dados
(SQLAlchemy)
│   │   ├── __init__.py
│   │   └── user.py         # Modelos de usuário, log, domínio, etc.
│   ├── routes/             # Definição das rotas da API (Blueprints)
│   │   ├── __init__.py
│   │   ├── auth.py         # Rotas de autenticação (login, registro, logout)
│   │   ├── user.py         # Rotas de gerenciamento de usuários e admin
│   │   └── ...             # Outras rotas (campanhas, templates, etc.)
│   ├── services/           # Lógica de negócios e integração com serviços
externos
│   │   ├── __init__.py
│   │   ├── cache_service.py
│   │   ├── telegram_service.py
│   │   └── ...
│   ├── security/           # Módulos de segurança
│   │   ├── __init__.py
│   │   ├── validators.py   # Validação de entrada e sanitização
│   │   ├── rate_limiter.py # Proteção contra força bruta
│   │   ├── two_factor_auth.py # Implementação de 2FA
│   │   ├── http_security.py # Headers de segurança HTTP
│   │   └── security_logger.py # Logging de eventos de segurança
│   └── utils/              # Funções utilitárias
├── tests/                  # Testes automatizados (unitários, integração)
├── instance/               # Arquivos de instância (e.g., banco de dados
SQLite)
├── logs/                   # Logs da aplicação
├── requirements.txt        # Dependências Python
├── .env.example            # Exemplo de variáveis de ambiente
└── ...
```

2.2. Componentes Chave do Backend

- **`main.py`**: Inicializa a aplicação Flask, configura extensões (SQLAlchemy, Bcrypt, CORS, etc.), registra Blueprints para as rotas e define o ponto de entrada.
- **`config.py`**: Centraliza as configurações da aplicação, permitindo diferentes ambientes (desenvolvimento, teste, produção) e carregamento de variáveis de ambiente.
- **`models/`**: Contém as classes que mapeiam as tabelas do banco de dados para objetos Python (ORM - SQLAlchemy). Define a estrutura dos dados e os relacionamentos entre eles.

- **`routes/`**: Organiza as rotas da API em Blueprints, que são componentes modulares do Flask. Cada Blueprint agrupa rotas relacionadas a uma funcionalidade específica (e.g., `auth.py` para autenticação, `user.py` para gerenciamento de usuários).
- **`services/`**: Abriga a lógica de negócios complexa e a integração com serviços externos. Por exemplo, `cache_service.py` para gerenciamento de cache, `telegram_service.py` para envio de notificações via Telegram.
- **`security/`**: Um módulo dedicado à implementação de funcionalidades de segurança, conforme detalhado na Fase 4. Inclui validação de entrada, rate limiting, 2FA, headers HTTP e logging de segurança.

2.3. Fluxo de Requisição no Backend

1. **Requisição HTTP**: O frontend (ou outro cliente) envia uma requisição HTTP para uma rota específica do backend.
2. **Middleware de Segurança**: A requisição passa por middlewares de segurança (e.g., rate limiting, verificação de headers de segurança) antes de chegar à rota.
3. **Autenticação/Autorização**: Se a rota exigir, o usuário é autenticado (verificando tokens JWT ou sessões) e suas permissões são verificadas para autorizar o acesso ao recurso.
4. **Validação de Entrada**: Os dados da requisição (JSON, formulário) são validados usando schemas (e.g., Marshmallow) e sanitizados para prevenir ataques como XSS e SQL Injection.
5. **Lógica de Negócios**: A rota chama funções nos módulos de `services/` para executar a lógica de negócios necessária (e.g., criar um usuário, iniciar uma campanha).
6. **Interação com DB**: Os serviços interagem com o banco de dados através dos modelos definidos em `models/` (SQLAlchemy ORM).
7. **Resposta HTTP**: O backend constrói uma resposta HTTP (geralmente JSON) com os dados solicitados ou mensagens de status e a envia de volta ao cliente.

3. Detalhamento do Frontend (React)

O frontend React é uma Single Page Application (SPA) que interage com o backend através de chamadas de API.

3.1. Estrutura de Diretórios

```
phishing-manager-frontend/
├── public/
│   └── index.html          # Ponto de entrada HTML
├── src/
│   ├── index.js            # Ponto de entrada React
│   ├── App.js              # Componente principal da aplicação
│   ├── components/         # Componentes reutilizáveis (botões, inputs, cards)
│   ├── pages/              # Páginas da aplicação (Login, Dashboard, Users,
Campaigns)
│   ├── services/           # Funções para interagir com a API do backend
│   │   └── auth.js         # Funções de autenticação
│   │   └── user.js         # Funções de usuário
│   ├── contexts/           # Context API para gerenciamento de estado global
│   ├── hooks/              # Hooks personalizados
│   ├── assets/             # Imagens, ícones, fontes
│   ├── styles/             # Estilos globais ou variáveis CSS
│   └── utils/              # Funções utilitárias
├── package.json            # Dependências Node.js e scripts
├── .env.development        # Variáveis de ambiente para desenvolvimento
├── .env.production         # Variáveis de ambiente para produção
└── ...
```

3.2. Componentes Chave do Frontend

- **`index.js`**: Renderiza o componente `App` no DOM.
- **`App.js`**: Define o roteamento da aplicação (usando `react-router-dom`) e pode incluir provedores de contexto globais.
- **`components/`**: Contém componentes React menores e reutilizáveis que podem ser usados em várias páginas (e.g., `Button`, `InputField`, `Table`).
- **`pages/`**: Cada arquivo nesta pasta representa uma

página específica da aplicação (e.g., `LoginPage`, `DashboardPage`, `UsersPage`). - **`services/`**: Contém funções que encapsulam as chamadas à API do backend. Isso centraliza a lógica de comunicação com a API e facilita a manutenção e o tratamento de erros. Por exemplo, `auth.js` pode ter funções para `login`, `register` e `logout`. - **`contexts/`**: Utiliza a Context API do React para gerenciar o estado global da aplicação, como informações do usuário logado, status de autenticação, ou configurações

globais. - **`hooks/`**: Contém hooks personalizados para encapsular lógica reutilizável entre componentes.

3.3. Fluxo de Interação no Frontend

1. **Navegação do Usuário**: O usuário interage com a interface, clicando em botões, preenchendo formulários ou navegando entre as páginas.
2. **Componentes e Estado**: Os componentes React reagem às interações do usuário, atualizando seu estado interno e, consequentemente, a interface.
3. **Chamadas à API**: Quando uma ação do usuário requer dados do backend (e.g., login, listar usuários, criar campanha), as funções dos módulos `services/` são chamadas para fazer requisições HTTP à API RESTful do backend.
4. **Gerenciamento de Estado Global**: Os dados recebidos do backend são processados e, se necessário, atualizam o estado global da aplicação através dos contextos React.
5. **Renderização da UI**: A interface do usuário é re-renderizada para refletir as mudanças no estado, apresentando os novos dados ou o feedback da operação ao usuário.

4. Interação entre Frontend e Backend

A comunicação entre o frontend e o backend é a espinha dorsal da aplicação, baseada em princípios RESTful e JSON para troca de dados.

4.1. API RESTful

O backend expõe uma API RESTful que o frontend consome. As operações comuns incluem: - **GET**: Para recuperar dados (e.g., `/api/users` para listar usuários). - **POST**: Para criar novos recursos (e.g., `/api/register` para registrar um novo usuário). - **PUT/PATCH**: Para atualizar recursos existentes (e.g., `/api/users/{id}` para atualizar um usuário). - **DELETE**: Para remover recursos (e.g., `/api/users/{id}` para deletar um usuário).

Todas as respostas da API são formatadas em JSON, facilitando o parsing e o uso no frontend.

4.2. Autenticação e Autorização

O sistema utiliza um mecanismo de autenticação baseado em tokens (e.g., JWT - JSON Web Tokens) ou sessões seguras para proteger as rotas da API. O fluxo típico é:

1. **Login**: O frontend envia as credenciais do usuário para a rota de login do backend.
2. **Geração de Token/Sessão**: O backend valida as credenciais e, se corretas, gera um token JWT (ou estabelece uma sessão) e o retorna ao frontend.
3. **Armazenamento do Token**: O frontend armazena o token (e.g., em `localStorage` ou cookies seguros) e o inclui em todas as requisições subsequentes para rotas protegidas (geralmente no cabeçalho `Authorization`).
4. **Verificação do Token**: O backend, ao receber uma requisição com um token, verifica sua validade e autenticidade. Se o token for válido, a requisição é processada; caso contrário, uma resposta de não autorizado (401 Unauthorized) é retornada.
5. **Autorização**: Além da autenticação, o backend também implementa lógica de autorização para garantir que o usuário autenticado tenha as permissões necessárias para acessar o recurso solicitado (e.g., apenas administradores podem gerenciar outros usuários).

4.3. Tratamento de Erros

O backend retorna códigos de status HTTP apropriados e mensagens de erro detalhadas em JSON para o frontend. O frontend, por sua vez, é responsável por interpretar esses erros e exibir mensagens amigáveis ao usuário, além de lidar com cenários como tokens expirados ou acesso negado.

5. Banco de Dados

O Phishing Manager utiliza um banco de dados relacional para armazenar seus dados. A escolha padrão para desenvolvimento e implantação simples é SQLite, devido à sua natureza de arquivo único e facilidade de uso. No entanto, a arquitetura é projetada para ser compatível com outros bancos de dados relacionais, como PostgreSQL ou MySQL, para ambientes de produção que exigem maior escalabilidade e robustez.

5.1. SQLAlchemy ORM

O backend utiliza o SQLAlchemy, um Object Relational Mapper (ORM) para Python. O ORM permite que os desenvolvedores interajam com o banco de dados usando objetos Python em vez de escrever SQL diretamente. Isso traz vários benefícios:

- **Abstração**: O código se torna independente do banco de dados subjacente, facilitando a troca de um banco de dados para outro.
- **Segurança**: Ajuda a prevenir ataques de SQL Injection, pois as consultas são construídas de forma segura pelo ORM.
- **Produtividade**: Reduz a quantidade de código necessário para interagir com o banco de dados.
- **Manutenibilidade**: Os modelos de dados são definidos em classes Python, tornando-os mais fáceis de entender e manter.

5.2. Modelos de Dados Principais

Os modelos de dados são definidos no diretório `src/models/` do backend. Alguns dos modelos principais incluem:

- **`User`**: Armazena informações sobre os usuários do sistema, incluindo credenciais, papéis (administrador, usuário comum), status (ativo, banido), e configurações de 2FA.
- **`Domain`**: Representa os domínios utilizados nas campanhas de phishing.
- **`UserDomain`**: Tabela de relacionamento entre usuários e domínios, indicando quais domínios um usuário pode gerenciar.
- **`Campaign`**: Detalhes sobre as campanhas de phishing, como nome, status, data de criação, etc.
- **`Template`**: Modelos de e-mail ou páginas de phishing reutilizáveis.
- **`GeneratedURL`**: URLs únicas geradas para cada vítima em uma campanha.
- **`Log`**: Registra eventos importantes do sistema, incluindo atividades do usuário, eventos de segurança, erros e auditoria.

5.3. Migrações de Banco de Dados

Para gerenciar as mudanças no esquema do banco de dados ao longo do tempo, é recomendável usar uma ferramenta de migração de banco de dados, como o Alembic (que se integra bem com o SQLAlchemy). Isso permite que as alterações no modelo de dados sejam aplicadas de forma controlada e reversível em diferentes ambientes.

6. Segurança na Arquitetura

A segurança é uma preocupação central no design do Phishing Manager. Diversas camadas de proteção foram implementadas para mitigar riscos e proteger os dados e a integridade do sistema.

6.1. Validação e Sanitização de Entrada

- **Marshmallow Schemas**: Todos os dados recebidos via API são validados rigorosamente usando schemas do Marshmallow. Isso garante que os dados estejam no formato e tipo esperados, prevenindo dados malformados ou maliciosos.
- **Sanitização XSS**: Campos de texto que podem conter conteúdo gerado pelo usuário são sanitizados para remover scripts maliciosos e tags HTML perigosas, prevenindo ataques de Cross-Site Scripting (XSS).
- **Prevenção de SQL Injection**: O uso do SQLAlchemy ORM protege automaticamente contra a maioria dos ataques de SQL Injection, pois as consultas são parametrizadas e não construídas diretamente com concatenação de strings.

6.2. Autenticação e Autorização

- **Senhas Fortes**: O sistema impõe políticas de senhas fortes, exigindo complexidade (maiúsculas, minúsculas, números, símbolos) e comprimento mínimo. As senhas são armazenadas de forma segura usando hashing (Bcrypt) e salting.
- **Autenticação de Dois Fatores (2FA)**: Para contas administrativas, o 2FA baseado em TOTP (Time-based One-Time Password) é obrigatório, adicionando uma camada extra de segurança contra acesso não autorizado.

- **Controle de Acesso Baseado em Papéis (RBAC)**: O sistema diferencia usuários comuns de administradores, e as rotas da API são protegidas para garantir que apenas usuários com as permissões adequadas possam acessá-las.

6.3. Proteção contra Ataques Comuns

- **CSRF (Cross-Site Request Forgery)**: Implementação de tokens CSRF para proteger contra requisições forjadas de outros sites.
- **Rate Limiting**: Utilização de `Flask-Limiter` para impor limites de requisições por IP ou por usuário, prevenindo ataques de força bruta, negação de serviço (DoS) e varredura de endpoints.
- **Headers de Segurança HTTP**: Configuração de headers como `Content-Security-Policy (CSP)`, `X-Content-Type-Options`, `X-Frame-Options`, `Strict-Transport-Security (HSTS)` e `Referrer-Policy` para mitigar diversos ataques baseados em navegador.
- **Logging de Segurança**: Eventos de segurança críticos (tentativas de login falhas, alterações de permissão, acesso a recursos sensíveis) são registrados em logs detalhados para auditoria e detecção de anomalias.

6.4. Boas Práticas de Desenvolvimento Seguro

- **Princípio do Menor Privilégio**: Componentes e usuários operam com o mínimo de privilégios necessários para realizar suas funções.
- **Atualização de Dependências**: Manutenção regular das dependências para garantir que vulnerabilidades conhecidas sejam corrigidas.
- **Tratamento de Exceções**: Captura e tratamento adequado de exceções para evitar vazamento de informações sensíveis em mensagens de erro.
- **Variáveis de Ambiente**: Uso de variáveis de ambiente para armazenar informações sensíveis (chaves secretas, credenciais de banco de dados) em vez de codificá-las no código fonte.

7. Observabilidade e Manutenção

Para garantir a estabilidade e facilitar a manutenção do Phishing Manager, a arquitetura incorpora mecanismos de observabilidade e práticas de manutenção.

7.1. Logging

O sistema utiliza um sistema de logging centralizado que registra eventos em diferentes níveis (DEBUG, INFO, WARNING, ERROR, CRITICAL). Os logs são essenciais para:

- **Depuração**: Identificar e resolver problemas no código.
- **Auditoria**: Rastrear atividades do usuário e eventos de segurança.
- **Monitoramento**: Acompanhar o desempenho e a saúde da aplicação.
- **Análise de Erros**: Entender a causa raiz de falhas e exceções.

Os logs podem ser configurados para serem gravados em arquivos, enviados para um serviço de log centralizado ou exibidos no console.

7.2. Testes Automatizados

Uma suíte abrangente de testes automatizados é fundamental para garantir a qualidade e a estabilidade do software. O Phishing Manager inclui:

- **Testes Unitários**: Verificam a funcionalidade de componentes individuais (funções, classes, modelos) isoladamente.
- **Testes de Integração**: Validam a comunicação entre diferentes módulos e serviços (e.g., backend e banco de dados, frontend e API).
- **Testes End-to-End (E2E)**: Simulam o fluxo completo do usuário através da aplicação, desde a interface até o backend e o banco de dados.
- **Testes de Segurança**: Validam a eficácia das medidas de segurança implementadas.

7.3. CI/CD (Integração Contínua / Entrega Contínua)

A configuração de um pipeline de CI/CD (e.g., com GitHub Actions) automatiza o processo de construção, teste e implantação da aplicação. Isso garante que cada alteração no código seja testada automaticamente, reduzindo a chance de introduzir bugs e acelerando o ciclo de desenvolvimento.

7.4. Monitoramento e Alertas

Em um ambiente de produção, é crucial monitorar a saúde e o desempenho da aplicação. Isso pode incluir:

- **Métricas de Desempenho**: Uso de CPU, memória, latência de requisições, taxa de erros.
- **Disponibilidade**: Verificação se a aplicação está online e respondendo.
- **Alertas**: Notificações automáticas para a equipe de operações em caso de anomalias ou falhas críticas.

8. Considerações Finais

A arquitetura do Phishing Manager é projetada para ser robusta, segura e escalável, fornecendo uma base sólida para o desenvolvimento contínuo e a adição de novas funcionalidades. A modularidade e a separação de preocupações garantem que o sistema possa evoluir sem comprometer sua estabilidade ou segurança.

Ao seguir os princípios e práticas descritos neste guia, os desenvolvedores podem contribuir de forma eficaz para o projeto, mantendo a alta qualidade e segurança do Phishing Manager.

Autor: Manus AI **Data:** 28 de Junho de 2025

Referências:

[1] Flask Documentation: https://flask.palletsprojects.com/en/latest/ [2] React Documentation: https://react.dev/ [3] SQLAlchemy Documentation: https://docs.sqlalchemy.org/en/latest/ [4] Marshmallow Documentation: https://marshmallow.readthedocs.io/en/stable/ [5] Flask-Limiter Documentation: https://flask-limiter.readthedocs.io/en/stable/ [6] PyOTP Documentation: https://pyotp.readthedocs.io/en/latest/ [7] Flask-Talisman Documentation: https://pypi.org/project/Flask-Talisman/ [8] OWASP Top 10: https://owasp.org/www-project-top-ten/ [9] GitHub Actions Documentation: https://docs.github.com/en/actions [10] Docker Documentation: https://docs.docker.com/ [11] Systemd Documentation: https://www.freedesktop.org/wiki/Software/systemd/