Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.

Argos
Documento de Arquitetura de Software

Versão <2.7>

Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.

Histórico da Revisão

Data	Versão	Descrição	Autor
20/08/2025	1.0	Elaboração da versão inicial do: 1.2. Escopo 1.4 Referências 10. Qualidade: - Portabilidade - Segurança	Nathalia G. Silva
20/08/2025	1.1	Elaboração da versão inicial do: 8. Visão de Dados (Opcional) 10. Qualidade: - Escalabilidade - Confiabilidade - Disponibilidade	Kennedy Rodrigo T. Gonçalves
25/08/2025	1.2	Revisão do item 10. Qualidade, correção da Disponibilidade	Nathalia G. Silva
03/09/2025	1.3	Elaboração da versão inicial do: 2. Requisitos e Restrições da Arquitetura: - Plataforma - Persistência - Internacionalização	Kennedy Rodrigo T. Gonçalves
03/09/2025	1.4	Elaboração da versão inicial do: 2. Requisitos e Restrições da Arquitetura: - Segurança - Linguagem	Nathalia G. Silva
17/09/2025	1.5	Elaboração da versão inicial do: 4. Visão Lógico 4.1. Visão Geral 4.2. Pacotes de Design	Kennedy Rodrigo T. Gonçalves
17/09/2025	1.6	Desenvolvimento do item: 8. Visão de dados (opcional)	Pedro Henrique O. Marques
18/09/2025	1.7	Elaboração do item: 3. Visão de Caso de Uso	Matheus Vinycius V. Batista
20/09/2025	1.8	Elaboração da versão inicial do: 6. Visão de Implantação 6.1. Visão Geral da Arquitetura de Implantação 6.2. Configuração de implantação de serviço	Nathalia G. Silva
23/09/2025	1.9	Desenvolvimento do item: 9. Volume e Desempenho	Pedro Henrique O. Marques
30/09/2025	2.0	Revisão do item 4: Adição do Diagrama de Sequência e detalhamento das camadas	Kennedy Rodrigo T. Gonçalves
01/10/2025	2.1	Desenvolvimento inicial do item 4: Adição da visão lógica do front-end, adição da descrição do fluxo	Gustavo Horeste S. Barros

Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.

03/10/2025	2.2	Revisão do item 4: Adição do diagrama de sequência da criação de usuário e desativação de conta de usuário.	Gustavo Horeste S. Barros
10/10/2025	2.3	Revisão do item 1: Elaboração da introdução, 1.1 finalidade, 1.3. Definições, Acrônimos e Abreviações.	Gustavo Horeste S. Barros
10/10/2025	2.4	Desenvolvimento do item 3. Visão de Casos de Uso/História de Usuários	Matheus Vinycius V. Batista
10/10/2025	2.5	Desenvolvimento do item 10. Plano de Interação	Kennedy Rodrigo T. Gonçalves
15/10/2025	2.6	Revisão dos Tópicos	Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.
30/10/2025	2.7	Adição do diagrama de implantação	Nathalia G.Silva

Índice Analítico

Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.

Documento de Arquitetura de Software

1. Introdução

Este documento tem como intuito sintetizar a arquitetura completa do software Argos. A solução foi estruturada em duas camadas principais: uma de Cliente (Front-end) e uma de Servidor (Back-end). A camada de Cliente foi desenvolvida utilizando React e Vite, adotando o padrão MVVM (Model-View-ViewModel) para privilegiar a criação de componentes desacoplados e de fácil manutenção. Já no back-end, utilizamos Python e o framework FastAPI para a construção de um server-side robusto e escalável, organizado sob uma Arquitetura em Camadas. Para a persistência de dados transacionais, utilizamos o MongoDB e SQLite, e o serviço Elasticsearch é empregado para a análise inteligente e eficiente de logs de alto volume.

1.1 Finalidade

Este documento oferece uma visão geral arquitetural abrangente do sistema Argos, usando diversas visões arquiteturais para representar diferentes aspectos do sistema. O objetivo deste documento é capturar e comunicar as decisões arquiteturais significativas que foram tomadas em relação ao sistema. O documento irá adotar uma estrutura baseada na visão "4+1" de modelo de arquitetura, conforme (Kruchten, 1994).

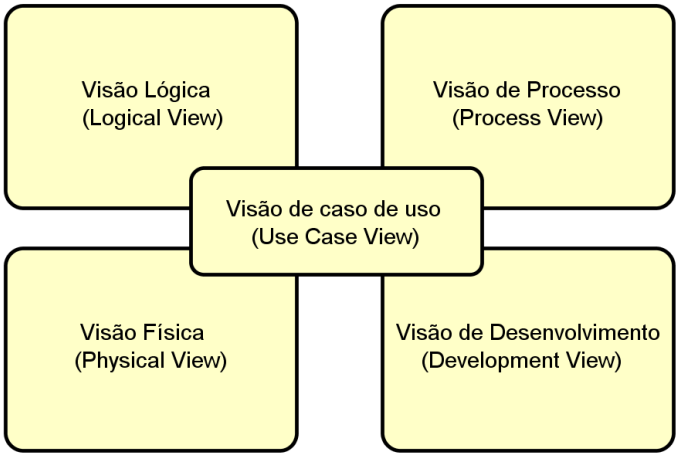


Figura 1 - Visões 4+1 da arquitetura de software (Kruchten, 1994)

1.2 Escopo

O sistema proposto é voltado para a área de análise forense digital, oferecendo suporte à detecção de atividades suspeitas, à condução de investigações de segurança e à geração de relatórios de conformidade prontos para auditorias. A solução pretende atender empresas com múltiplos clientes, equipes especialmente de SOC (Security Operations Center), fornecendo uma interface de monitoramento e controle interativo de logs baseada em regras heurísticas e integração com blacklists de IPs e domínios maliciosos.

1.3 Definições, Acrônimos e Abreviações

DBA:	Administrador de Banco de Dados.
DTO:	Objeto para transferência de dados entre camadas (equivalente aos nossos Schemas).

Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.

FastAPI:	Framework web utilizado para a construção da API REST do back-end.
JWT:	Padrão para criação de tokens de acesso.
ORM:	Mapeamento Objeto-Relacional (SQLAlchemy).
Pydantic:	Biblioteca para validação de dados (usada nos Schemas).
REST:	Estilo arquitetural para sistemas distribuídos.
SOC:	Centro de Operações de Segurança.
SQLAlchemy:	Biblioteca ORM utilizada para interação com o banco de dados.
MVVM	Arquitetura Model, View, ViewModel.

1.4 Referências

A seguir estão os documentos desenvolvido anteriormente

1. *Documento de Visão* - Criado em 08/05/2025
2. *Documento de Requisitos de Software* - Criado em 21/05/2025
3. *Repositório do Projeto* - Criado em 7/08/2025

2. Requisitos e Restrições da Arquitetura

Requisito	*Solução*
Linguagem	**Python** 3.11+ para o desenvolvimento do back-end, e **REACT**, **VITE** e **TAILWIND**
Plataforma	**FastAPI** como framework web para a API REST, **SQLite** como sistema de banco de dados para persistência e **React+** no front-end
Segurança	**JWT:** Autenticação em tokens protegendo os endpoints; Senhas criptografadas em hash; RBAC: controle de acesso baseado em perfis
Persistência	**SQLite** para o armazenamento de dados relacionais da aplicação gerenciado pelo ORM **SQLAlchemy;** **Elasticsearch** para a ingestão, indexação, armazenamento e análise de grandes volumes de dados de log, atendendo aos requisitos de monitoramento em tempo real e consultas complexas

Contêinerização e Implantação	**Docker** será utilizado para conteinerizar a aplicação e seus serviços, garantindo portabilidade entre diferentes ambientes

Tabela 1 – Exemplo de requisitos e restrições

3. Visão de Casos de Uso/Histórias de Usuário

Será abordado nesta seção as histórias de usuário centrais que devem estar no MVP do sistema.

HU1 - Incluir um novo usuário

Descrição: COMO gestor de uma área SOC QUERO incluir um novo usuário, PARA que ele utilize o sistema de acordo com seu Perfil de Acesso E receba as credenciais por e-mail.

HU7 - Criar Perfil de Permissão

Descrição: COMO gestor de uma área SOC QUERO criar perfis PARA atribuir as permissões apropriadas a cada usuário ao sistema E manter os dados contidos no sistema Argos protegidos.

HU8 - Atribuir perfil ao usuário

Descrição: COMO gestor de uma área SOC QUERO atribuir um perfil a um usuário PARA QUE cada usuário tenha acesso apenas às informações relevantes E evite acesso indevido a dados sensíveis

HU13 - Realizar monitoramento em tempo real de logs de acesso

Descrição: COMO um usuário QUERO monitorar tentativas de login(bem-sucedidas e falhas) dos funcionários que fizeram login, endereços IP de origem, timestamps, comandos executados, PARA detectar tentativas de acesso não autorizado, escalonamento de privilégios e atividades suspeitas de usuários comprometidos.

HU14. Configurar ponto de monitoramento

Descrição: COMO um usuário QUERO configurar um servidor da empresa como um ponto de monitoramento dedicado, especificando os parâmetros de coleta e centralização dos logs de acesso ao sistema operacional de outros servidores PARA que eu possa consolidar os dados de log em um local seguro, facilitando a coleta e transporte.

HU16. Configurar alerta de atividades suspeitas

Descrição: COMO usuário QUERO configurar regras de alerta personalizadas para serem notificadas sobre atividades suspeitas detectadas nos logs de acesso PARA identificar proativamente potenciais ameaças à segurança, investigá-las rapidamente e tomar as medidas corretivas necessárias.

4. Visão Lógica

Esta seção descreve as partes mais significativas da arquitetura do modelo de design do sistema Argos, focando em sua decomposição em pacotes e classes, e nas responsabilidades de cada componente, além de demonstrar também através de diagramas de sequência como a comunicação entre as camadas ocorre.

4.1 Visão Geral

Esta subseção descreve toda a decomposição do modelo de design arquitetural para back-end e front-end.

Back-end (Serviços):

A decomposição do modelo de design do back end do sistema foi realizada utilizando uma Arquitetura em Camadas (Layered Architecture). Esta arquitetura assegura a separação de responsabilidades, onde cada camada possui uma função definida.

Dessa forma separamos em:

- **Camada de API/Rotas:** A camada mais externa, responsável por expor os endpoints RESTful, receber as

requisições HTTP, validar dados de entrada com os schemas e aplicar as dependências de segurança.

- **Camada de Controle (controllers)**: Atua como uma ponte entre a camada de API e a de Serviço. Sua única responsabilidade é repassar a requisição ao serviço apropriado.
- **Camada de Serviço (services):** Atua como orquestrador da lógica de negócio. A lógica irá permanecer nesta camada quando precisa:
  - Coordenar múltiplos objetos de domínio (ex: criar um Usuário e associar um Perfil).
  - Utilizar Repositórios para buscar ou salvar diferentes entidades
  - Interagir com sistemas externos (ex: enviar um e-mail)
- **Camada de Repositório (repositories):** A camada de acesso a dados. É a única camada que interage diretamente com o banco de dados, abstraindo as consultas e manipulações de dados.
- **Camada de Modelo (model):** Define a estrutura das tabelas com SQLAlchemy e também contêm a lógica de negócio inerente ao próprio objeto.
- **Schemas**: define os contratos de dados da API usados para validação e serialização.

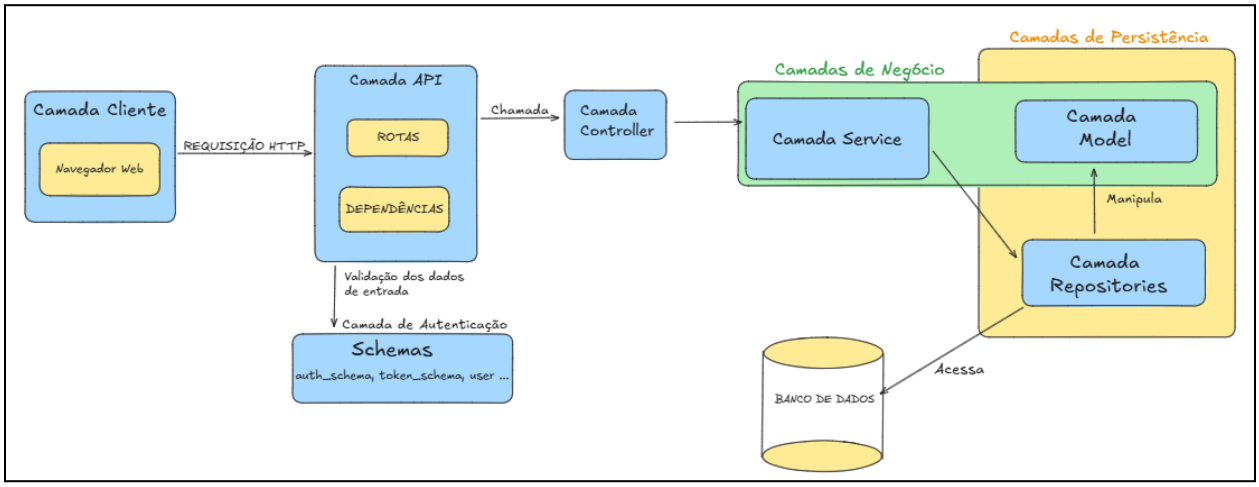


Figura 2 - Exemplo de diagrama de camadas da aplicação

Para evitar os extremos de um "domínio anêmico" (onde a camada “model” serve apenas para estruturar dados) ou um "domínio rico" (onde a “model” contém toda a lógica), foi adotada uma abordagem de domínio equilibrado. Nessa abordagem, a lógica de negócio é distribuída de forma inteligente entre a Camada de “Service” e a Camada “Model”. A seguir temos a um diagrama que demonstra a lógica utilizada para a separação da lógica de negócio entre as camadas model e service:

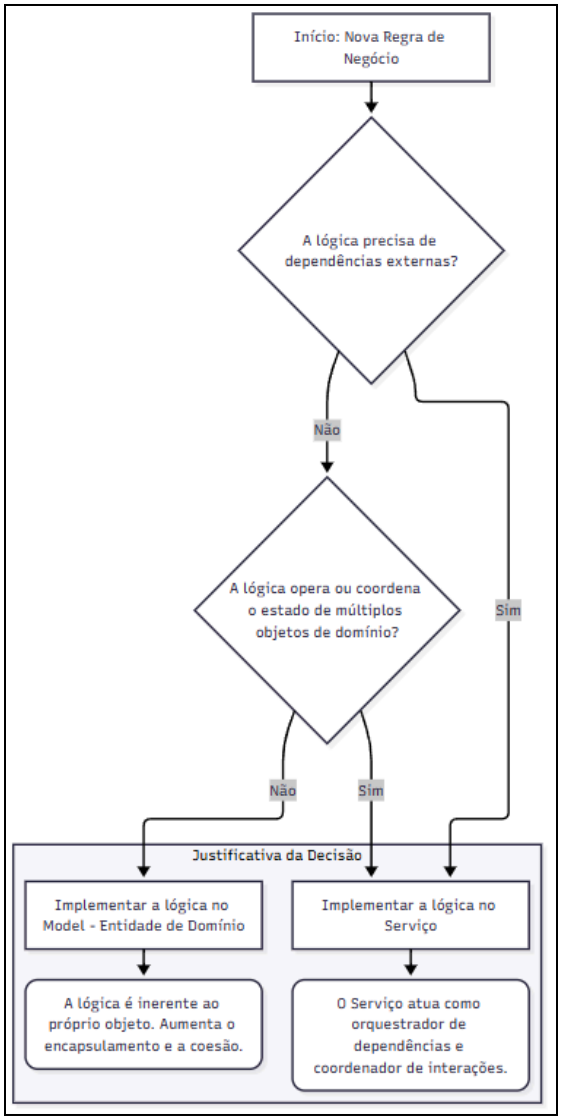


Figura 3 - Diagrama da lógica utilizada para separar as regras de negócio entre a camada model e service

Front-end (Camada de Apresentação):

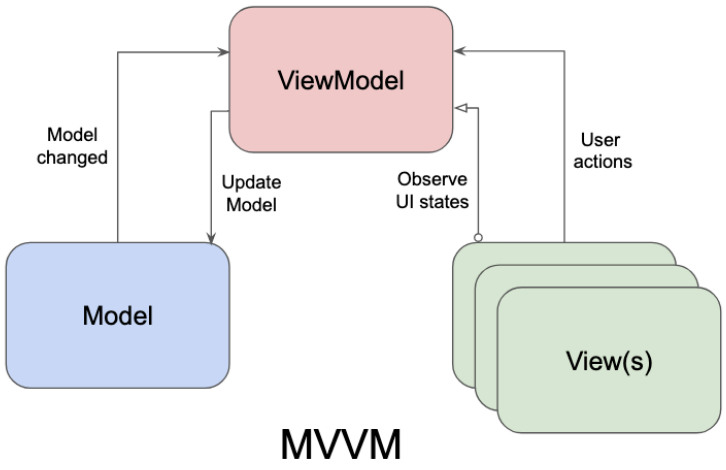
Já a decomposição do modelo de design do front end do sistema foi realizada utilizando a Arquitetura MVVM (Model-View-ViewModel). Essa arquitetura promove a desacoplagem entre a interface de usuário e a lógica de apresentação, favorecendo a testabilidade, a escalabilidade e a manutenção do código.

Dessa forma, separamos em:

- **Camada de View (React Components):** Responsável pela renderização da interface e interação direta com o usuário. Os componentes recebem dados do ViewModel e exibem de forma reativa, reagindo a mudanças de estado.
- **Camada de ViewModel:** Atua como intermediária entre a View e o Model. É responsável por manter o estado da aplicação, aplicar transformações necessárias nos dados recebidos do Model e expor esses dados prontos para a View. No React, isso pode ser implementado utilizando hooks personalizados e contextos

para centralizar a lógica de apresentação.

- **Camada de Model:** Representa a fonte de dados da aplicação. Pode englobar chamadas à API, acesso a repositórios locais (como LocalStorage ou IndexedDB) ou estruturas de dados internas. O Model não conhece a View, sendo acessado e manipulado apenas pelo ViewModel.



4.2 Interação dinâmica e colaboração entre os pacotes

A seção anterior descreveu as responsabilidades estáticas de cada pacote na arquitetura em camadas do sistema Argos. Esta seção focará em ilustrar algumas interação dinâmica e a colaboração entre esses pacotes através da análise de fluxos de trabalho de casos significativos do projeto.

4.2.1 Fluxo de Criação de Usuário

Este fluxo é realizado apenas por um gestor autenticado. Ele percorre todas as camadas para a criação do usuário, fornecendo ao final do processo o e-mail institucional e senha gerada de forma aleatória para o funcionário. Uma vez que a criação/desativação do usuário é função do gestor, entretanto, apenas o funcionário tem acesso a sua conta.

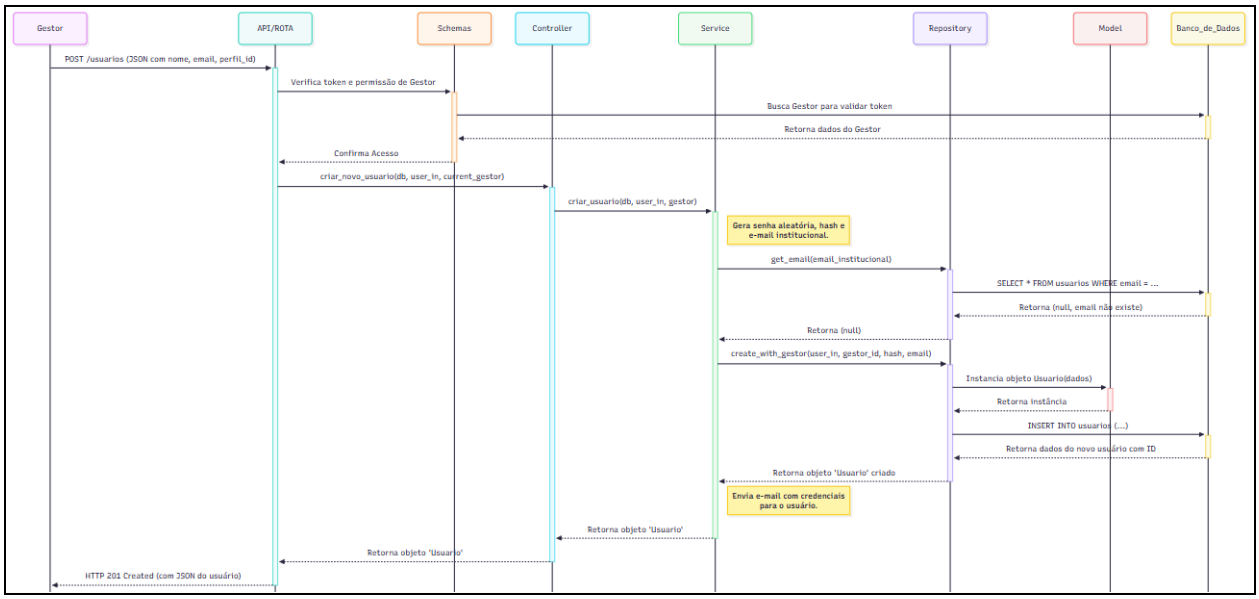


Figura 4 - Diagrama de Sequência - Criação de Usuário

Descrição do Fluxo - Back end:

1. **Requisição e Segurança (Camada de API):**
   - Um cliente autenticado como **Gestor** envia uma requisição POST para o endpoint /usuarios.
   - O **Roteador** (user_router) recebe a requisição. Antes de prosseguir, ele aciona a dependência "**verificar_token"**, que valida o token JWT do gestor para garantir que ele tem permissão para realizar a ação.
2. **Delegação (Controller):**
   - Após a confirmação do acesso, o Roteador chama o método correspondente no **Controlador** (user_controller).
   - O Controlador, atuando como uma ponte, imediatamente delega a tarefa para a **Camada de Serviço**, passando os dados do novo usuário e o objeto do gestor logado.
3. **Lógica de Negócio (Service):**
   - O **Serviço** (user_service), que é o orquestrador da lógica de negócio, executa as seguintes ações:
     - Gera as credenciais do novo usuário (senha aleatória, hash da senha e e-mail institucional).
     - Realiza uma validação de negócio: antes de criar o usuário, ele solicita ao **Repositório** que verifique se o e-mail institucional gerado já existe no banco de dados.
     - Recebendo a confirmação de que o e-mail é único, o Serviço comanda o Repositório a prosseguir com a criação.
4. **Acesso a Dados e Persistência (Repositório e Model):**
   - O **Repositório** (user_repository) recebe os dados do serviço.
   - Ele instancia um objeto do **Model** Usuario, preenchendo-o com os dados validados (incluindo a senha criptografada).
   - O Repositório executa o comando INSERT no **Banco de Dados**.
   - Após a confirmação da criação, o Repositório retorna o objeto Usuario completo (agora com um ID) para o Serviço.
5. **Ação Externa (Serviço):**
   - Com o usuário criado, o Serviço executa a ação de enviar um e-mail com as credenciais para o novo usuário.

6. **Resposta:**
    - O objeto Usuário é retornado em “cascata” pelas camadas (Service -> Controller-> API/ROTA).
    - O FastAPI serializa o objeto usando o UserResponseSchema e envia uma resposta “HTTP 201 Created” para o cliente.

Descrição do Fluxo - Front end:

1. **Interação do Usuário (Camada View)** O componente de Interface (View) recebe a entrada de dados do usuário e captura o evento de submissão do formulário, que é delegado para a camada ViewModel.
2. **Início da Lógica (Camada ViewModel)** O Módulo de Lógica (ViewModel) é acionado.
    - O Módulo de Lógica estabelece o estado de carregamento para `true`.
    - Executa a validação de formato e obrigatoriedade dos dados de entrada localmente.
    - Se a validação for bem-sucedida, o **Módulo de Lógica** delega a requisição ao **Módulo de Serviços de Rede**.
3. **Comunicação com a API (Camada Networking/Storage Service)** O Módulo de Serviços de Rede (Service) é acionado.
    - O **Módulo de Serviços** recebe os dados do usuário.
    - O **Módulo de Serviços** envia a requisição `POST` para o endpoint `/usuarios` da API.
    - **Segurança (Implícita):** O cliente HTTP anexa o token JWT, permitindo que o *back-end* verifique a permissão do usuário (espelhando a validação de token do Back-end).
    - O Módulo de Serviços recebe a resposta do *back-end* (HTTP 201 Created ou um erro).
4. **Resposta e Tratamento (Camada Networking/Storage e ViewModel)** O Módulo de Serviços repassa o resultado da requisição ao Módulo de Lógica.
    - **Em caso de sucesso (HTTP 201):** O **Módulo de Lógica** limpa o estado do formulário e prepara uma notificação de sucesso.
    - **Em caso de erro (4xx):** O **Módulo de Lógica** captura a mensagem de erro detalhada do *back-end* (Ex: regra de negócio, como "E-mail já existe") e a armazena no estado de erros.
    - O **Módulo de Lógica** seta o estado de carregamento para `false`.
    - A **Interface (View)** reage automaticamente, exibindo o resultado da operação ao usuário.

4.2.2 Fluxo de Criação de Perfil

Este fluxo é realizado apenas por um gestor autenticado. O processo garante que um perfil só seja criado se as permissões associadas a ele forem válidas.

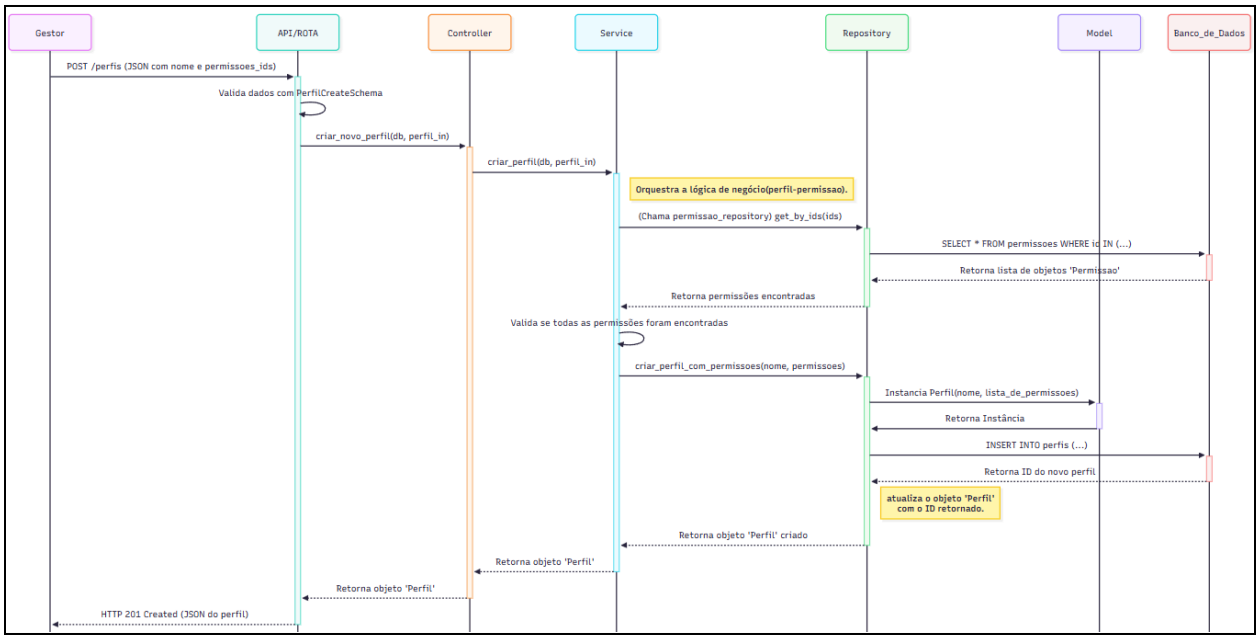


Figura 4 - Diagrama de Sequência - Criação de Perfil

Descrição do Fluxo - Backend:

1. **Requisição e Validação (Camada de API):**
   - Um Gestor envia uma requisição POST para o endpoint /perfis, contendo um JSON com o nome do perfil e uma lista de IDs de permissões.
   - O **Roteador** (perfil_router) recebe a requisição e, antes de prosseguir, o FastAPI utiliza o "PerfilCreateSchema" para validar automaticamente a estrutura e os tipos dos dados recebidos.
2. **Delegação (Controller):**
   - Com os dados validados, o Roteador invoca o método correspondente no Controlador (perfil_controller).
   - O Controlador, agindo como uma ponte, repassa imediatamente a chamada para a Camada de Serviço.
3. **Lógica de Negócio (Service):**
   - O **Serviço** (perfil_service) orquestra a lógica de negócio. Sua primeira tarefa é validar a existência das permissões solicitadas. Para isso, ele chama o "permissao_repository" para buscar no banco de dados todos os objetos "Permissao" correspondentes aos IDs recebidos.
   - O Serviço então executa a regra de negócio: ele verifica se a quantidade de permissões encontradas é a mesma que a quantidade de IDs solicitados. Se não for, uma exceção é lançada.
   - Com as permissões validadas, o Serviço comanda o **Repositório** de perfis (perfil_repository) para criar o novo perfil, passando o nome e a lista de objetos de permissão.
4. **Persistência (Repositório e Model):**
   - O **Repositório** recebe os dados do serviço. Ele então instancia o objeto "Perfil" na camada Model, associando a ele os objetos de Permissao.
   - O Repositório utiliza este objeto Perfil para executar a operação de INSERT no **Banco de Dados**.
   - Após o banco confirmar a criação e retornar o novo ID, o SQLAlchemy atualiza a instância do Model em memória com esse ID.
   - O Repositório retorna o objeto Perfil completo e persistido para o Serviço.
5. **Resposta:**

- O objeto Perfil é retornado em “cascata” pelas camadas (Service -> Controller-> API/ROTA).
- Finalmente, o FastAPI serializa o objeto usando o PerfilResponseSchema e envia uma resposta “HTTP 201 Created ao Cliente, contendo” os dados do perfil recém-criado.

Descrição do Fluxo - Front end:

1. **Interação do Usuário (Camada View)** O componente de Interface (View), responsável pelo formulário de criação de perfil, recebe o nome do perfil e a lista de IDs de permissões selecionadas. O evento de submissão do formulário é delegado para a camada ViewModel.
2. **Início da Lógica (Camada ViewModel)** O Módulo de Lógica (ViewModel) é acionado.
   - O Módulo de Lógica estabelece o estado de carregamento para **true**.
   - Executa a validação local (Ex: verificação de campo "nome do perfil" obrigatório).
   - Se a validação local for bem-sucedida, o **Módulo de Lógica** delega a requisição ao **Módulo de Serviços de Rede**, passando o nome e a lista de IDs de permissões.
3. **Comunicação com a API (Camada Networking/Storage Service)** O Módulo de Serviços de Rede (Service), responsável pela comunicação HTTP, é acionado.
   - O **Módulo de Serviços** recebe o objeto com nome e IDs de permissões.
   - O **Módulo de Serviços** envia a requisição POST para o endpoint /perfis da API.
   - O cliente HTTP (por exemplo, Axios) garante que o corpo da requisição esteja formatado em JSON, espelhando o **Schema** do *back-end* (Garantindo que a estrutura e tipos atendam ao PerfilCreateSchema)
4. **Resposta e Tratamento (Camada Networking/Storage e ViewModel)**
   - O *back-end* executa sua lógica (validação de unicidade, verificação da existência de todas as permissões) e retorna a resposta (HTTP 201 Created ou erro).
   - O Módulo de Serviços recebe a resposta e a repassa ao Módulo de Lógica.
5. **Feedback Final (Camada ViewModel e View)**
   - **Em caso de sucesso (HTTP 201):** O **Módulo de Lógica** prepara uma notificação de sucesso e, opcionalmente, limpa o estado do formulário de criação.
   - **Em caso de erro (4xx):** O **Módulo de Lógica** captura a mensagem de erro do *back-end* (Ex: "Permissão não encontrada" ou "Nome de perfil já existe") e a armazena no estado de erros.
   - O **Módulo de Lógica** seta o estado de carregamento para false.
   - A **Interface (View)** reage automaticamente, exibindo a mensagem de sucesso ou o erro (Tailwind) ao usuário.

4.2.3 Fluxo de Desativação da conta de um Usuário

Este fluxo é realizado apenas por um gestor autenticado. Apenas o gestor é capaz de realizar tanto a ativação quanto a desativação da conta do usuário. A lógica de negócio é dividida: o “Service” orquestra a operação, mas a regra de como alterar o estado do usuário pertence ao próprio objeto de domínio na camada “Model”.

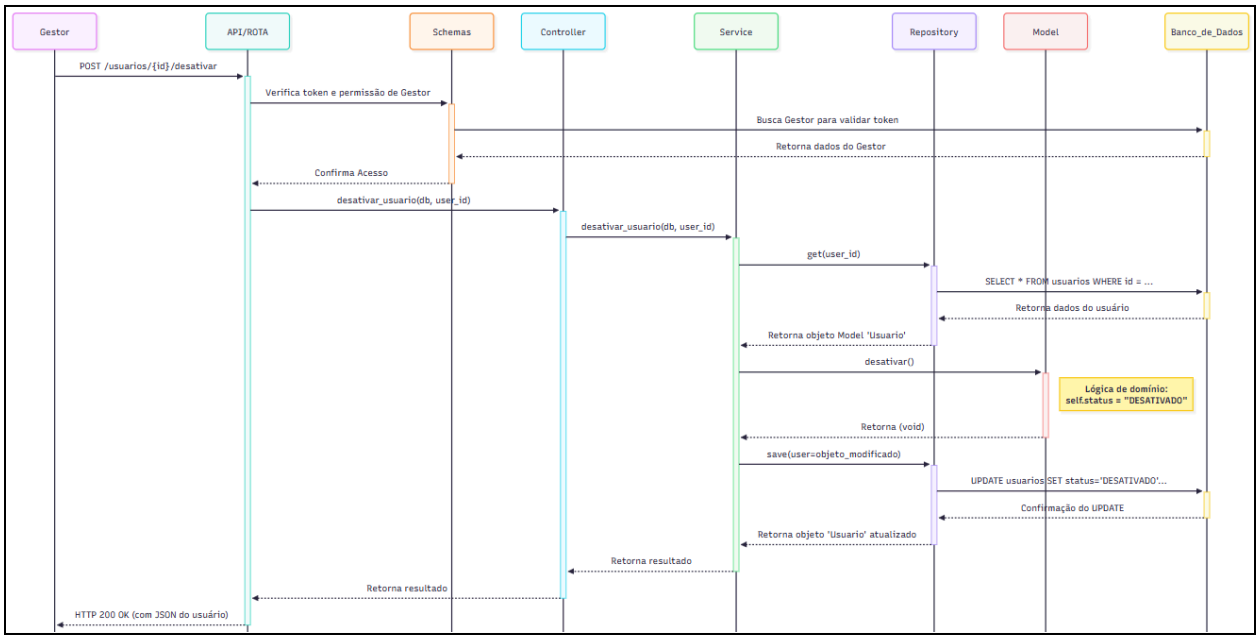


Figura 5 - Diagrama de Sequência - Desativação da conta de Usuario

Descrição do Fluxo - Backend:

1. **Requisição e Segurança (Camada de API):**
   - Um Gestor envia uma requisição POST para o endpoint "../{id}/desativar" para inativar um usuário específico.
   - O **Roteador** (user_router) recebe a requisição. Antes de prosseguir, ele aciona a dependência "**verificar_token"**, que valida o token JWT do gestor para garantir que ele tem permissão para realizar a ação.
2. **Delegação em Camadas:**
   - Com o acesso confirmado, o Roteador chama o método correspondente no Controlador (user_controller).
   - O Controlador repassa a chamada para a Camada de Serviço, que é responsável por executar a lógica.
3. **Orquestração do Serviço:**
   - O Serviço (user_service) inicia o processo. Sua primeira tarefa é obter o objeto de usuário que precisa ser modificado. Para isso, ele solicita ao Repositório (user_repository) que busque o usuário pelo ID fornecido.
4. **Acesso a Dados (Repositório):**
   - O Repositório executa uma query SELECT no Banco de Dados para encontrar o usuário.
   - O banco retorna os dados do usuário, e o Repository retorna o objeto Usuario da Model para o Service
5. **Execução da Lógica de Domínio (Model):**
   - Em vez de o Service manipular o estado diretamente, ele delega essa responsabilidade ao próprio objeto na camada Model. (domínio equilibrado)
   - O Serviço chama o método desativar() no objeto Usuario retornado.
   - É o próprio Model que encapsula a regra de como alterar seu status interno.
6. **Persistência da Alteração:**
   - Após o método desativar() modificar o objeto em memória, o Serviço entrega este objeto

modificado de volta ao Repositório, com o comando para salvar as alterações.
- O Repositório, através do SQLAlchemy, traduz essa ação em um comando UPDATE no Banco de Dados, persistindo a mudança de status.

7. **Resposta:**
   - O Repositório retorna o objeto Usuario atualizado para o Serviço.
   - O objeto é então retornado em cascata pelas camadas (Service -> Controller-> API/ROTA).
   - Finalmente, o FastAPI serializa o objeto e envia uma resposta HTTP 200 OK ao Cliente, confirmando que o usuário foi desativado com sucesso.

Descrição do Fluxo - front end:

1. **Interação do Usuário (Camada View)** A interface (View) onde o usuário logado (Gestor) visualiza a lista de usuários exibe um botão ou opção "Desativar" para um usuário específico.
   - O Gestor clica no botão "Desativar", acionando um evento.
   - O evento chama a função de comando correspondente no Módulo de Lógica (ViewModel), passando o id do usuário a ser desativado.
2. **Início da Lógica (Camada ViewModel)** O Módulo de Lógica (ViewModel) recebe o comando e o id.
   - O **Módulo de Lógica** estabelece o estado de carregamento específico (ou global) para true.
   - O **Módulo de Lógica** delega a requisição de comando ao **Módulo de Serviços de Rede**, passando o id do usuário.
3. **Comunicação com a API (Camada Networking/Storage Service)** O Módulo de Serviços de Rede (Service) é acionado.
   - O **Módulo de Serviços** constrói e envia uma requisição POST para o endpoint de comando /usuarios/{id}/desativar, usando o id fornecido.
   - **Segurança (Implícita):** O cliente HTTP anexa o token JWT do Gestor, permitindo que o *back-end* execute a dependência de "verificar_token" para validar a permissão (espelhando a Camada de API).
4. **Resposta e Tratamento (Camada Networking/Storage e ViewModel)**
   - O *back-end* executa a lógica (busca o usuário, chama o método desativar() no objeto Model e persiste a alteração) e retorna uma resposta de sucesso (Ex: HTTP 200 OK ou 204 No Content) ou erro.
   - O Módulo de Serviços recebe a resposta e a repassa ao Módulo de Lógica.
5. **Feedback Final e Atualização da View (Camada ViewModel e View)**
   - **Em caso de sucesso (200/204):** O **Módulo de Lógica** atualiza o estado local de dados (a lista de usuários) para refletir a desativação do usuário (ex: removendo-o da lista ativa ou alterando seu status visualmente).
   - **Em caso de erro (4xx):** O **Módulo de Lógica** captura a mensagem de erro e a armazena no estado para exibição.
   - O **Módulo de Lógica** seta o estado de carregamento para false.
   - A **Interface (View)** reage à atualização do estado de usuários, mostrando a desativação bem-sucedida.

5. Visão de Implantação

Esta seção descreve a configuração da rede física (hardware) e a disposição dos componentes de software na qual o Argos é implantado e executado. A arquitetura foi concebida para ser flexível, utilizando um modelo como serviço (SaaS), para garantir a portabilidade e a consistência entre diferentes ambientes.

5.1 Visão Geral da Arquitetura de Implantação

A implantação do Argos é baseada em um conjunto de serviços que se comunicam através de interfaces bem definidas. Os principais componentes lógicos a serem implantados são:

- **Agentes de Coleta:** Componentes leves instalados nos servidores do cliente a serem monitorados. Sua função é coletar e encaminhar logs de acesso para o núcleo do sistema.

- **Núcleo Argos:** Um conjunto de serviços que executam as centrais da aplicação:
  - ➢ **Serviço de API (Back-end):** Aplicação FastAPI , expõe os endpoints RESTful e implementa a lógica de negócio nas camadas de Serviço, Repositório e Modelo.
  - ➢ **Análise e Armazenamento de Logs:** Um contêiner executa o Elasticsearch, responsável pela ingestão, indexação e análise de grandes volumes de dados de log, sendo crucial para o monitoramento em tempo real e consultas complexas.
  - ➢ **Banco de Dados Relacional:** O SQLite, que gerencia o armazenamento dos dados relacionais da aplicação, como usuários, perfis e permissões, é gerenciado pelo ORM SQLAlchemy.
  - ➢ **Interface do Usuário (Front-end):** Uma aplicação web (React) que se comunica com o Serviço de API via requisições HTTPS.

5.2 Configuração de Implantação como Serviço

O Núcleo Argos é hospedado em uma infraestrutura de nuvem, e o cliente é responsável apenas pelos Agentes e pelo acesso dos usuários.

1. **Nós Físicos (Computadores, CPUs):**
   - **Infraestrutura do Cliente:**
     - ➢ **Servidores Monitorados:** Nós na rede do cliente que executam os agentes de coleta.
     - ➢ **Estações de Trabalho dos Analistas:** Computadores que acessam a plataforma Argos através de um navegador web padrão pela Internet.

2. **Infraestrutura de Nuvem (Hospedagem Argos):**
   - **Plataforma de Aplicação:** A infraestrutura de nuvem executa o núcleo Argos como um conjunto de serviços interconectados. Isso inclui:
     - ➢ **Serviço de API (Back-end, FastAPI)**, que atua como o ponto de entrada para todos os dados e requisições.
     - ➢ **Análise e Armazenamento de Logs (Elasticsearch)** para processamento e consulta de logs.
     - ➢ **Banco de Dados Relacional (SQLite)** para dados da aplicação.
     - ➢ **Serviço Web:** A **Interface do Usuário (React)** é hospedada e servida diretamente da nuvem para o navegador dos analistas.

3. **Interconexões (Rede):**
   - **Agentes para a Nuvem:** Os agentes de coleta nos servidores do cliente estabelecem uma conexão segura (via HTTPS/TLS) pela Internet (WAN) para enviar os logs diretamente ao endpoint do serviço de API na nuvem.
   - **Usuários para a Nuvem:** Os analistas acessam a interface do usuário por meio de seus navegadores, conectando-se via HTTPS pela Internet à plataforma hospedada na nuvem. Toda a comunicação é criptografada e a autenticação de cada requisição é garantida por tokens JWT, assegurando que apenas usuários autorizados tenham acesso.
   - **Comunicação Interna na Nuvem:** A comunicação entre os componentes do Núcleo Argos (API, Elasticsearch, SQLite) ocorre dentro da rede segura da infraestrutura de nuvem, de forma otimizada para desempenho e segurança.

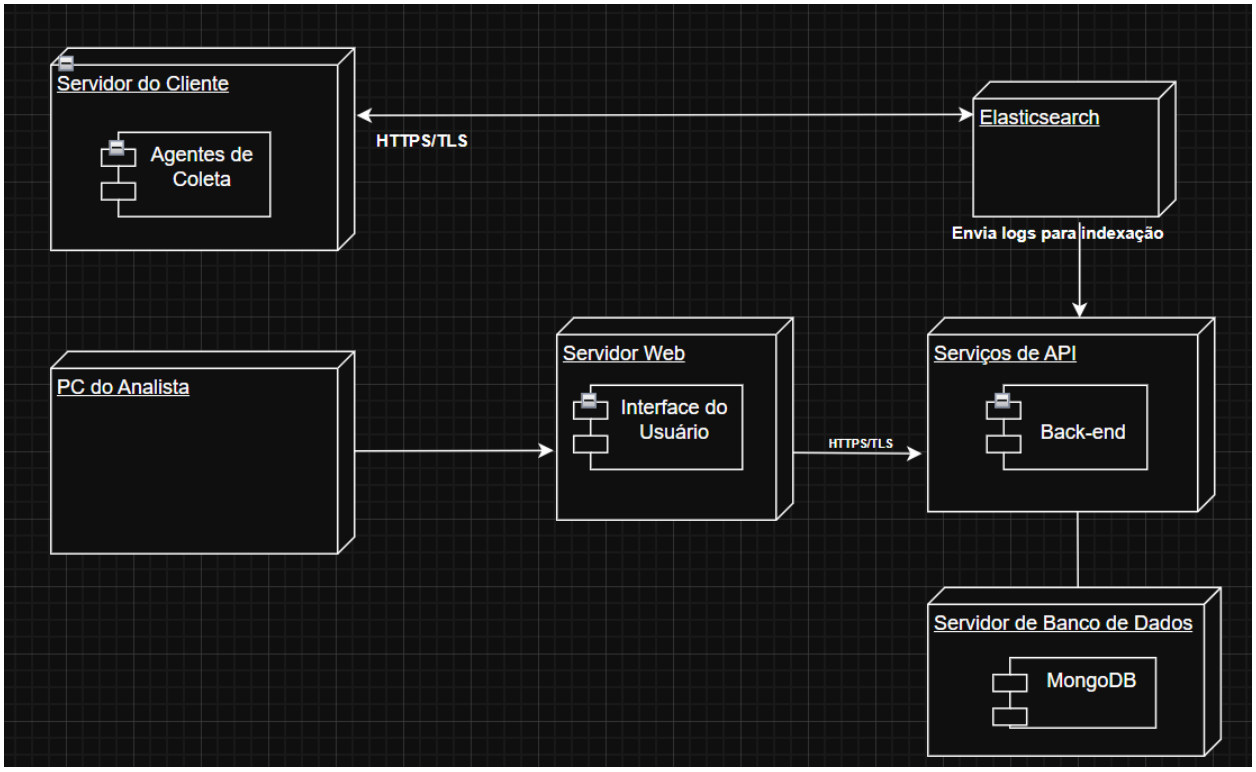
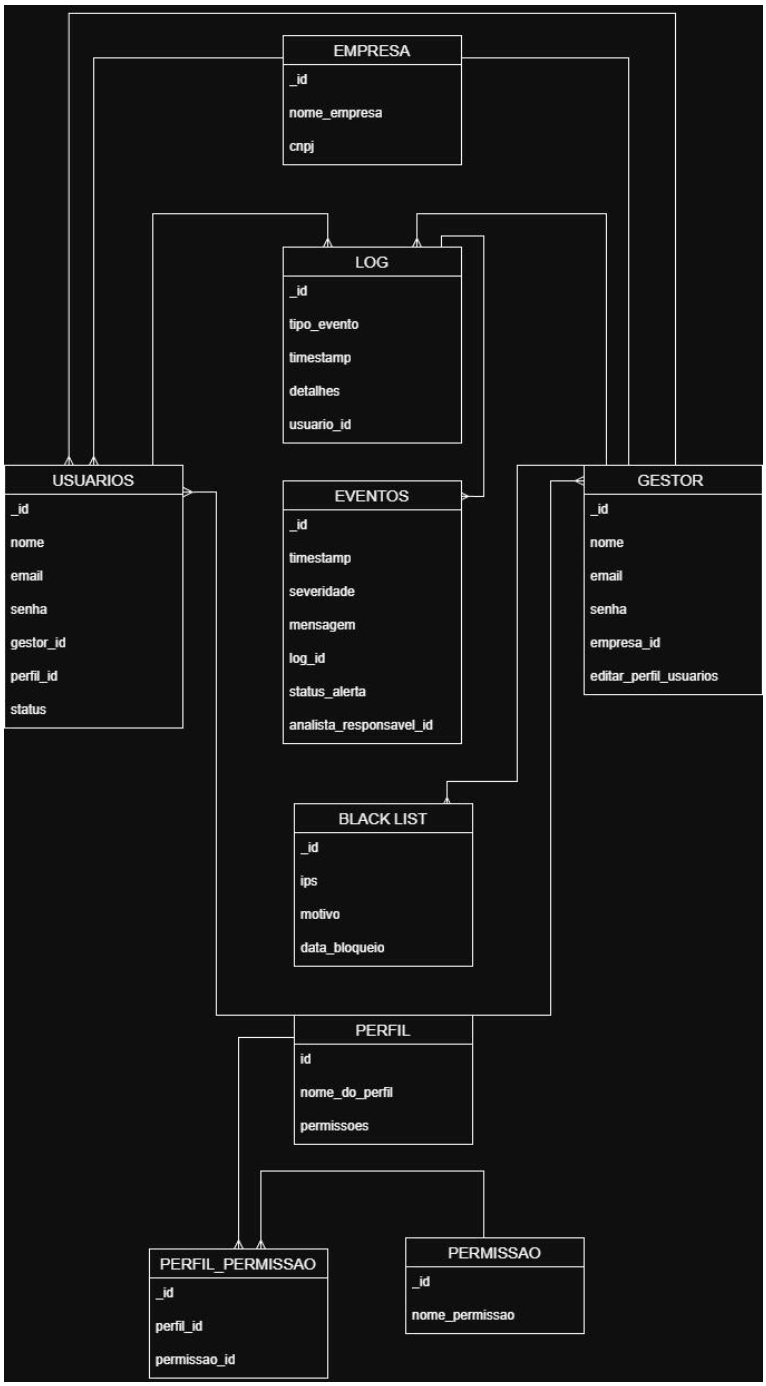


Figura 6 - Diagrama de Implantação

6. Visão de Dados

O modelo de dados do sistema Argos, detalhado na documentação técnica e ilustrado no Diagrama de Entidade-Relacionamento (DER), adota uma abordagem não relacional e flexível utilizando o MongoDB. A estrutura é composta por sete coleções principais, com relações estabelecidas por referências (_id). As entidades centrais são EMPRESA , que se relaciona um-para-um com o GESTOR e um-para-muitos com USUARIOS. Tanto

Gestores quanto Usuários geram registros na coleção LOG , que armazena detalhes e timestamps dos eventos para rastreabilidade. O controle de acesso é gerenciado através das coleções PERFIL e PERMISSAO , que se conectam em uma relação muitos-para-muitos por meio da coleção associativa PERFIL_PERMISSAO. Essa modelagem visa a escalabilidade e a eficiência na gestão de acessos e na coleta de logs.



1

Figura 1 - DER do Banco

7. Volume e Desempenho

Esta seção descreve as principais características de dimensionamento (volume) e as restrições de desempenho desejadas para o sistema Argos, conforme detalhado nos requisitos não funcionais. O sistema deve ser projetado para operar eficientemente sob as seguintes condições.

7.1. Volume (Capacidade e Escalabilidade)

- **Volume de Ingestão de Logs:** O sistema deve processar logs em tempo real com um volume de entrada de até **1 milhão de novos registros de log por hora**.
- **Capacidade de Dados Históricos:** O sistema deve ser capaz de armazenar logs brutos históricos por um período mínimo de **1 ano**.
- **Volume Máximo Anual:** O armazenamento deve acomodar um volume máximo anual de até **10 Terabytes** de dados brutos.
- **Escalabilidade Horizontal:** O sistema deve ser capaz de escalar horizontalmente para suportar um aumento de **50% no volume de logs** ingeridos por hora e um aumento de **40% no número de usuários concorrentes** dentro de 7 meses após o lançamento.

7.2. Desempenho (Latência e Tempo de Resposta)

Os seguintes requisitos de desempenho (latência/tempo de resposta) são fundamentais para garantir a eficácia da análise de segurança:

- **Latência de Visualização de Dados (RNF 001):** O sistema deve apresentar visualizações de dados na interface (dashboards, gráficos) com uma latência inferior a **2 segundos** para 95% das interações.
- **Latência de Processamento em Tempo Real (RNF 002):** O processamento de análises de logs em tempo real deve ter uma latência inferior a **5 segundos** (para 90% dos eventos), desde a ingestão do log até a disponibilização da análise para alerta ou visualização.
- **Tempo de Resposta para Consultas Analíticas Complexas (RNF 006):** O tempo de resposta para consultas analíticas complexas (que envolvem agregação de múltiplos dados para dashboards/relatórios) não deve exceder **30 segundos** (para 90% das consultas), considerando um volume de dados correspondente aos últimos 30 dias de ingestão.
- **Atualização do Painel de Ameaças:** Os dados de contagem de ameaças por severidade no painel devem ser atualizados automaticamente a cada **5 minutos** para refletir o estado mais recente possível.

7.3. Implicações Arquiteturais de Desempenho e Tolerância

A arquitetura de software deve endereçar os requisitos de volume e desempenho através das seguintes estratégias, conforme definido na Visão Lógica e nos Requisitos de Confiabilidade:

- **Processamento Distribuído:** O uso do **Elasticsearch** é mandatório para a ingestão, indexação, armazenamento e análise de grandes volumes de logs, atendendo aos requisitos de monitoramento em tempo real e consultas complexas, sendo a base para o cumprimento do RNF 002 e RNF 006.
- **Otimização de Consultas:** Consultas e *queries* devem ser otimizadas para garantir que relatórios e análises complexas (RNF 006) sejam gerados dentro de um tempo aceitável.
- **Ponto de Recuperação Objetivo (RPO):** Em caso de falha crítica, a perda máxima de dados (RPO) aceitável é de **15 minutos** de logs ingeridos antes da falha, garantindo a integridade dos dados forenses.
- **Tempo de Recuperação Objetivo (RTO):** Em caso de falha em um componente crítico, o sistema deve ser

capaz de se recuperar para o status operacional completo em até **1 hora**.

8. Qualidade

Esta seção descreve como a arquitetura de software contribui para os recursos do sistema

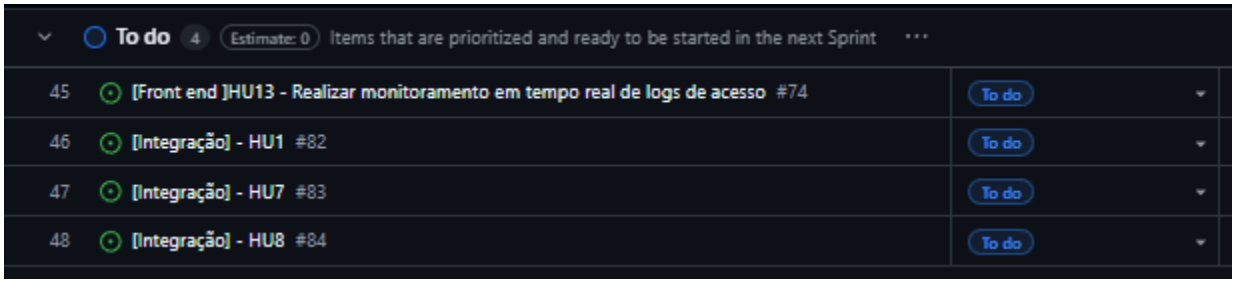
Item	*Descrição*	*Solução*
Escalabilidade	1. Comunicação Stateless : O servidor não armazena nenhuma informação sobre a sessão do cliente. Cada requisição contém toda a informação necessária para ser processada. 2. O backend e o frontend são desacoplados e evoluem de forma independente.	Conforme o volume de logs a serem analisados cresce, podemos simplesmente adicionar mais instâncias do servidor (escalabilidade horizontal) atrás de um load balancer. Como nenhuma instância guarda estado, qualquer uma pode atender a qualquer requisição, seja de um analista consultando um dashboard ou de um sistema enviando logs.
Confiabilidade	1. Endpoints de Verificação de Saúde (Health Checks).	Se uma instância do Argos perder a conexão com o banco de dados, o endpoint /health falhará.
Disponibilidade	1. A arquitetura deve suportar a implantação de novas versões de software e correções de segurança com o menor tempo de inatividade possível.	Garantir que as janelas de manutenção sejam curtas e previsíveis, mantendo o sistema acessível na maior parte do tempo.
Portabilidade	1. Interface Uniforme e Padrões Abertos: REST utiliza padrões universais da web, como o protocolo HTTP/S e formatos de dados como JSON. A interface (endpoints) é consistente e previsível.	O backend a ser desenvolvido em FastAPI, pode ser implantado em qualquer ambiente que suporte Python (Linux, Windows Server, containers Docker). O frontend será implementado através do Flutter que é capaz de gerar aplicativos para diversas plataformas com um único código base.
Segurança	1. Padrões Web: A comunicação REST opera sobre HTTPS, o que permite o uso de TLS/SSL para criptografar os dados em trânsito. A autenticação stateless é feita por requisição (Tokens JWT).	A criptografia via HTTPS é obrigatória para proteger a confidencialidade e a integridade das informações. A autenticação por token JWT a cada requisição garante que cada acesso seja

		verificado, o que é crucial para o controle de acesso baseado em perfis e para a proteção contra vazamento de Informações Pessoais Identificáveis.

9. Plano de Interação

1. Link do Kanban no Github: [Backlog · Fluxo de trabalho]

2. Print Geral do Kanban



Argos - Analisador Forense	**Grupo:** Gustavo Horestee S. Barros, Kennedy Rodrigo T. Gonçalves, Matheus Vinycius V. Batista, Nathalia G. Silva e Pedro Henrique O. Marques.