---

title: "Lab Reviewer API" date: "2024-03-15" summary: "This technical lab outlines the creation of a blog reviewing application using FastAPI and Langchain. The application transforms raw Markdown notes into structured blog posts, extracts and reviews code examples, translates content into Brazilian Portuguese, and generates PDFs of the output." tags: ["lab-reviewer", "fastapi", "langchain", "markdown", "code-examples", "translation", "pdf-generation"] published: true

---

API de Revisão de Blog

Visão Geral

Este laboratório técnico descreve a criação de uma aplicação de revisão de blog usando FastAPI e Langchain. A aplicação transforma notas brutas em Markdown em postagens de blog estruturadas, extrai e revisa exemplos de código, traduz o conteúdo para o português brasileiro e gera PDFs da saída.

Introdução

Como engenheiro de software, meu objetivo era aprimorar minhas habilidades e compartilhar minhas experiências por meio de um site de laboratório. Reconhecendo o potencial das tecnologias de IA, como FastAPI e Langchain, desenvolvi uma ferramenta assistida por IA para refinar meu conteúdo e traduzi-lo para o português. Esta aplicação organiza notas brutas em postagens estruturadas em Markdown, gera traduções e fornece revisões detalhadas do conteúdo.

Configuração do Projeto

Dependências

Para configurar o projeto, certifique-se de que as seguintes dependências estejam instaladas:

- **FastAPI**: `fastapi==0.110.0`
- **Uvicorn**: `uvicorn[standard]==0.29.0`
- **Variáveis de Ambiente**: `python-dotenv==1.0.1`
- **Validação de Dados**: `pydantic==2.6.4`
- **Manipulação de Formulários Multipart**: `python-multipart==0.0.9`

Stack LLM

- **OpenAI**: `openai==1.14.3`
- **Langchain**: `langchain==0.1.16`
- **Langchain-OpenAI**: `langchain-openai==0.1.3`
- **Langchain-Groq**: `langchain-groq`

Ferramentas Adicionais

- **Cliente HTTP Assíncrono**: `httpx==0.27.0`
- **Processamento de Markdown**: `markdown==3.6`
- **Geração de PDF**: `weasyprint==62.3`, `pydyf==0.10.0`

Configuração do Ambiente

1. Crie um ambiente virtual e ative-o:

   ```
   bash python -m venv venv source venv/bin/activate
   ```

2. Instale os pacotes necessários:

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

3. Copie o arquivo de ambiente de exemplo e configure suas variáveis de ambiente:

   ```
   bash cp .env.example .env
   ```

Edite o `.env` com suas chaves e valores de tempo de execução.

Funcionalidade Principal

A aplicação consiste em cinco agentes principais, cada um com responsabilidades específicas:

1. Agente de Escrita de Postagem de Lab

Este agente organiza notas brutas em postagens de lab coerentes e estruturadas, garantindo clareza e coesão.

```
class LabPostWriterAgent:
    """Agente responsável por transformar notas esboçadas em postagens est

    def __init__(self, llm: BaseChatModel | None = None) -> None:
        """Inicializa o modelo de chat usado pelo agente."""
        self.logger = logging.getLogger(__name__)
        self.agent_name = AgentRole.POST_WRITER
        self.llm = llm or LLMConfig.build_chat_model_for_agent(AgentRole.P
        self.blog_reviewer = LabReviewerAgent()
        self.code_example_agent = LabCodeExampleAgent()

    def organize_notes(self, request: LabPostWriterRequest) -> LabPostWrit
        """Transforma notas brutas em uma postagem de blog revisada em mar
        self.logger.info("agent=%s | iniciando o pipeline de organize_note
        enriched_context = enrich_context_with_repositories(request.contex
        examples_response = self.code_example_agent.extract_examples(
            LabCodeExampleRequest(
                notes_context=request.context,
```

```
                max_examples=3,
            )
        )

        code_examples_context = self._build_code_examples_context(examples
        final_context = enriched_context
        if code_examples_context:
            final_context = f"{enriched_context}\n\n{code_examples_context
        system_prompt = LabPostWriterPrompt.build_system_prompt()
        messages = [
            SystemMessage(content=system_prompt),
            HumanMessage(content=final_context),
        ]

        response = self.llm.invoke(messages)
        current_markdown = str(getattr(response, "content", "")).strip() o

        # Executa 3 ciclos completos de revisão/melhoramento usando os Age
        for iteration in range(1, 4):
            self.logger.info("agent=%s | ciclo %s/3 - solicitando revisão"
            revised = self.blog_reviewer.revise(LabReviewerRequest(content
            improvement_prompt = self._build_improvement_prompt(current_ma
            improved_response = self.llm.invoke([SystemMessage(content=sys
            current_markdown = str(getattr(improved_response, "content", "

        self.logger.info("agent=%s | pipeline finalizado (final_chars=%s)"
        return LabPostWriterResponse(reviewed_markdown=current_markdown)

    def _build_improvement_prompt(self, current_markdown, revised):
        return (
            "Você está melhorando uma postagem de blog após revisão editor
            "Aplique todas as correções e sugestões relevantes enquanto pr
            "Postagem atual:\n"
            f"{current_markdown}\n\n"
            "Versão revisada pelo editor:\n"
            f"{revised.revised_post}\n\n"
            "Erros encontrados:\n"
            + "\n".join(f"- {item}" for item in revised.errors_found)
            + "\n\nDicas de melhoria:\n"
            + "\n".join(f"- {item}" for item in revised.improvement_tips)
            + "\n\nChecklist da próxima revisão:\n"
            + "\n".join(f"- {item}" for item in revised.next_revision_chec
            + "\n\nRetorne apenas a postagem final melhorada em Markdown."
        )
```

2. Agente de Revisão de Lab

Este agente revisa postagens de lab, sugerindo melhorias para aumentar a qualidade e a legibilidade do conteúdo.

```
class LabReviewerAgent:
    """Agente responsável por revisar postagens de blog em Markdown."""
```

```
    def __init__(self, llm: BaseChatModel | None = None) -> None:
        self.logger = logging.getLogger(__name__)
        self.agent_name = AgentRole.REVIEWER
        self.llm = llm or LLMConfig.build_chat_model_for_agent(AgentRole.R

    def revise(self, request: LabReviewerRequest) -> LabReviewerResponse:
        messages = [
            SystemMessage(content=LabReviewerPrompt.build_system_prompt())
            HumanMessage(content=request.content),
        ]

        response = self.llm.invoke(messages)
        return LabReviewerResponse(
            revised_post=str(getattr(response, "revised_post", request.con
            errors_found=[],
            improvement_tips=[],
            next_revision_checklist=[]
        )
```

3. Agente de Exemplos de Código de Lab

Este agente extrai e revisa exemplos de código do GitHub através de uma integração de API, fornecendo exemplos para o Agente de Escrita de Postagem de Lab.

```
class LabCodeExampleAgent:
    """Extrai exemplos práticos de código de repositórios mencionados nas

    def extract_examples(self, request: LabCodeExampleRequest) -> LabCodeE
        repositories = self.github_provider.extract_repositories(request.n
        if not repositories:
            return LabCodeExampleResponse(examples=[], summary="Nenhum rep

        repo_context_sections = []
        for repository in repositories:
            try:
                repo_context_sections.append(self.github_provider.fetch_re
            except Exception as e:
                self.logger.error(f"Erro ao buscar contexto para {reposito

        messages = [
            SystemMessage(content=LabCodeExamplePrompt.build_system_prompt
            HumanMessage(content=self._format_human_context(request, repos
        ]

        response = self.llm.invoke(messages)
        return LabCodeExampleResponse(examples=response.get("examples", [])
```

4. Agente de Revisão de Metadados de Lab

Este agente gera um título e um resumo para o lab que podem ser usados no site do lab.

```
class LabPostMetadataAgent:
    """Gera metadados de frontmatter a partir do conteúdo revisado em mark

    def generate(self, request: LabPostMetadataRequest) -> LabPostMetadata
        messages = [
            SystemMessage(content=LabPostMetadataPrompt.build_system_promp
            HumanMessage(content=request.content),
        ]

        response = self.llm.invoke(messages)
        return LabPostMetadataResponse(
            title=str(getattr(response, "title", "")),
            date=str(getattr(response, "date", "")),
            summary=str(getattr(response, "summary", "")),
            tags=getattr(response, "tags", []),
            published=getattr(response, "published", True)
        )
```

5. Agente de Tradução de Postagem de Lab

Este agente traduz a postagem de lab revisada para o português brasileiro.

```
class LabPostTranslatorAgent:
    """Agente responsável por traduzir postagens revisadas para pt-BR."""

    def translate(self, request: LabPostTranslatorRequest) -> LabPostTrans
        messages = [
            SystemMessage(content=LabPostTranslatorPrompt.build_system_pro
            HumanMessage(content=request.content),
        ]

        response = self.llm.invoke(messages)
        translated_markdown = str(getattr(response, "content", "")).strip(
        return LabPostTranslatorResponse(translated_markdown=translated_ma
```

Fluxo de Processamento

1. **Escrevendo Conteúdo**: Os usuários inserem suas notas brutas em Markdown.
2. **Obter Exemplos de Código**: O Agente de Exemplos de Código de Lab extrai exemplos de código para melhorar a postagem do blog.
3. **Processo de Revisão**: O Agente de Escrita de Postagem de Lab organiza e revisa as notas através de três ciclos iterativos usando o Agente de Revisão de Blog.
4. **Processo de Metadados**: O Agente de Revisão de Metadados de Lab extrai metadados para o lab.

5. **Tradução**: O Agente de Tradução de Postagem de Lab converte o conteúdo finalizado em português.
6. **Saída**: A aplicação gera dois arquivos Markdown:
7. `your-roots-are-not-controllers_reviewed_pt_br.md`: versão traduzida
8. `your-roots-are-not-controllers_reviewed.md`: versão em inglês

Registro

A aplicação utiliza Langsmith para registro detalhado das interações dos agentes, que podem ser exploradas através de uma interface web. Além disso, um sistema de registro personalizado captura interações com os LLMs durante o processamento para melhorar a depuração e o monitoramento.

Desafios e Soluções

Desafios

1. Garantir que o agente revisor fornecesse feedback construtivo foi desafiador. O desempenho variou com diferentes modelos; tentativas iniciais com Grok e Meta-Llama produziram saídas menos coerentes, enquanto o GPT-4 da OpenAI melhorou significativamente a qualidade das revisões.
2. Permitir que cada agente utilizasse um tipo diferente de LLM foi necessário para otimizar o uso de tokens e reduzir custos.
3. O Agente de Exemplos de Código de Lab enfrentou dificuldades em extrair código do GitHub, exigindo múltiplas iterações de prompt para alcançar resultados satisfatórios.

Testes e Depuração

Para estabilidade e confiabilidade, testes unitários para a funcionalidade de cada agente devem ser escritos utilizando frameworks como `pytest`. O registro ajuda na resolução de problemas durante o desenvolvimento.

Melhorias Futuras

Extensões potenciais para a aplicação incluem:

- Implementação de uma interface amigável para upload de arquivos.
- Expansão do suporte a idiomas para maior acessibilidade.
- Desacoplamento da tarefa de fundo atual em uma arquitetura baseada em fila para melhor escalabilidade.
- Notificação aos usuários via email quando o processamento estiver completo.
- Adição de um Agente de Criação de Imagem para melhorar a atratividade visual.

Conclusão

O revisor de lab ajuda com sucesso na aprendizagem e compartilhamento de conteúdo com qualidade e acessibilidade aprimoradas. Ao aproveitar a IA através do FastAPI e Langchain, esta aplicação apoia a criação e revisão de conteúdo enquanto facilita o suporte multilíngue, expandindo o alcance das obras escritas.

Para o código completo e mais detalhes, visite o repositório.

Contexto do Repositório

Estrutura do Projeto

- **`main.py`**: Ponto de entrada da aplicação e middleware.
- **`blog/router.py`**: Rotas da API (`/blog-post-writer/*`).
- **`blog/service.py`**: Orquestração para agentes de escrita, tradução e revisão.
- **`blog/agents/`**: Agentes e esquemas específicos de funcionalidades.
- **`public/markdowns/`**: Arquivos markdown de saída gerados.
- **`core/`**: Configuração compartilhada e configuração do LLM.

Executar Localmente

Inicie a aplicação com:

```
uvicorn main:app --reload --host 0.0.0.0 --port 3015
```

Verifique a saúde da aplicação:

```
curl http://127.0.0.1:3015/
```

Endpoints da API

`POST /blog-post-writer/organize-notes`

Aceita um upload `.md` em UTF-8 (`file`). O processamento ocorre em segundo plano.

Exemplo:

```
curl -X POST http://127.0.0.1:3015/blog-post-writer/organize-notes \
  -F "file=@notes.md"
```

Retorna o caminho de saída para o arquivo revisado. O serviço também grava um arquivo traduzido com o sufixo `_pt_br`.

`GET /outputs/markdown`

Lista arquivos markdown gerados disponíveis em `public/markdowns`.

Exemplo:

```
curl http://127.0.0.1:3015/outputs/markdown
```

Docker

Construa e execute a aplicação usando Docker:

```
docker build -t blog-reviewer-app .
docker run --env-file .env -p 3015:80 blog-reviewer-app
```

Notas

- Apenas arquivos `.md` são aceitos em `organize-notes`.
- Nomes dos arquivos de saída são derivados do nome do arquivo enviado e gravados em `public/markdowns/`.
- Postagens geradas sempre incluem frontmatter de metadados em YAML com campos obrigatórios: `title`, `date`, `summary`, `tags` e `published`.
- Não comite o arquivo `.env` ou chaves de API reais.