Api

1 API Reference - FACIN_IA

1.1 Visão Geral

Este documento contém a referência completa das funções, classes e módulos disponíveis no projeto FACIN_IA.

1.2 Módulo Principal: `app.py`

1.2.1 Importações Principais

```
from langchain_groq import ChatGroq
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, END, START
from langchain.tools import tool
```

1.2.2 Classes

1.2.2.1 `AgentState(TypedDict)`

Estado compartilhado entre agentes no workflow LangGraph.

```
class AgentState(TypedDict):
    messages: Annotated[List[BaseMessage], operator.add]
```

Atributos: - `messages`: Lista de mensagens (HumanMessage, AIMessage, ToolMessage) com operador de agregação

Tipos de Mensagem: - `HumanMessage`: Mensagens do usuário - `AIMessage`: Respostas dos agentes (Groq/OpenAI) - `ToolMessage`: Resultados de execução de ferramentas

---

1.2.3 Ferramentas (@tool)

1.2.3.1 `query_folha_database(sql_query: str) -> str`

Executa consultas SQL **SOMENTE SELECT** no banco de dados de Folha de Pagamento.

Parâmetros: - `sql_query` (str): Consulta SQL SELECT válida

Retorna: - `str`: Resultados formatados em tabela (até 15 linhas)

Tabelas Disponíveis:

```
tb_servidores (id, nome, cpf, matricula, orgao, cargo)
tb_folha_pagamento (id, matricula, competencia, vencimentos, descontos, liquido)
```

Exemplo:

```
result = query_folha_database(
    "SELECT nome, cargo FROM tb_servidores WHERE orgao = 'Secretaria da Saúde'"
)
```

Segurança: - Apenas SELECT permitido - Rejeita UPDATE, DELETE, INSERT, DROP - Validação antes da execução

1.2.4 Nós do Grafo

1.2.4.1 `groq_agent_node(state: AgentState) -> dict`

Nó do agente Groq (Llama 3.1) para consultas de folha de pagamento.

Modelo: `llama-3.1-8b-instant`
Temperatura: 0.2

Parâmetros: - `state` (AgentState): Estado atual com histórico de mensagens

Retorna: - `dict`: `{"messages": [AIMessage]}` com resposta do agente

Ferramentas: `query_folha_database`

1.2.4.2 `openai_agent_node(state: AgentState) -> dict`

Nó do agente OpenAI (GPT-3.5) para consultas de folha de pagamento.

Modelo: `gpt-3.5-turbo`
Temperatura: 0.2

Parâmetros: - `state` (AgentState): Estado atual com histórico de mensagens

Retorna: - `dict`: `{"messages": [AIMessage]}` com resposta do agente

Ferramentas: `query_folha_database`

1.2.4.3 `route_junction_node(state: AgentState) -> dict`

Nó de junção/roteamento (hub central do grafo).

Função: Atua como ponto central de decisão sem modificar o estado.

1.2.4.4 `router_logic(state: AgentState) -> str`

Lógica de roteamento condicional que decide o próximo nó.

Parâmetros: - `state` (AgentState): Estado atual

Retorna: - `"tools"`: Se AIMessage contém tool_calls - `"groq_agent"`: Se @groq mencionado ou alternância par - `"openai_agent"`: Se @openai mencionado ou alternância ímpar - `"__end__"`: Se resposta final sem tool_calls

Lógica de Alternância:

```
# Conta mensagens AI
ai_count = sum(1 for msg in messages if isinstance(msg, AIMessage))

# Alternância
if ai_count % 2 == 0:
    return "groq_agent"  # Par → Groq
else:
    return "openai_agent"  # Ímpar → OpenAI
```

Menções Explícitas: - `@groq` → força roteamento para Groq - `@openai` → força roteamento para OpenAI

1.2.4.5 `compila_grafo() -> CompiledGraph`

Compila o grafo de estados LangGraph com todos os nós e arestas.

Retorna: - `CompiledGraph`: Aplicativo executável do workflow

Estrutura do Grafo:

```
START → router → [groq_agent | openai_agent | tools | END]
                      ↓              ↓            ↓
                  router ← ————————————————————————
```

Nós: - `router`: Nó de junção - `groq_agent`: Agente Groq/Llama3 - `openai_agent`: Agente OpenAI/GPT - `tools`: Executor de ferramentas

Arestas: - START → router (sempre) - router → {groq_agent, openai_agent, tools, END} (condicional) - groq_agent → router (sempre) - openai_agent → router (sempre) - tools → router

(sempre)

Uso:

```
app = compila_grafo()
result = app.invoke({"messages": [HumanMessage(content="Quantos servidores ativos?")]})
```

1.3 Módulo: `cria_db.py`

1.3.1 Constantes

```
DB_FILE = "folha_pagamento.db"
SQL_FILE = "criacao_banco.sql"
```

1.3.2 Funções

1.3.2.1 `cria_database() -> tuple[sqlite3.Connection, sqlite3.Cursor]`

Cria ou recria o banco de dados SQLite executando script SQL.

Retorna: - `tuple`: (conexão, cursor) ou (None, None) em caso de erro

Comportamento: 1. Remove banco existente (se houver) 2. Cria novo banco conectando ao SQLite 3. Executa script `criacao_banco.sql` 4. Retorna conexão e cursor

Exemplo:

```
conn, cursor = cria_database()
if conn:
    print("Banco criado com sucesso")
```

1.3.2.2 `popula_tabelas(conn: sqlite3.Connection, cursor: sqlite3.Cursor) -> None`

Popula as tabelas com dados do arquivo CSV.

Parâmetros: - `conn` (sqlite3.Connection): Conexão com o banco - `cursor` (sqlite3.Cursor): Cursor para execução SQL

Fonte de Dados: - `folha_pe_200linhas.csv` → 200 registros de exemplo

Tabelas Populadas: 1. `tb_servidores`: Dados únicos de servidores (nome, cpf, matrícula, orgão, cargo) 2. `tb_folha_pagamento`: Dados de folha (matrícula, competência, vencimentos, descontos, líquido)

Arquivos Gerados: - `servidores.xlsx`, `servidores.csv` - `folha.xlsx`, `folha.csv`

1.3.2.3 `main() -> None`

Função principal que orquestra a criação e população do banco.

Fluxo:

```
1. Verifica se DB_FILE existe
2. Chama cria_database()
```

3. Chama popula_tabelas()
4. Fecha conexão

1.4 LangChain Integrations

1.4.1 Chat Models

1.4.1.1 `ChatGroq`

Modelo de linguagem do Groq.

```
llm_groq = ChatGroq(
    model="mixtral-8x7b-32768",
    temperature=0.7,
    api_key=os.getenv("GROQ_API_KEY")
)
```

1.4.1.2 `ChatOpenAI`

Modelo de linguagem da OpenAI.

```
llm_openai = ChatOpenAI(
    model="gpt-4",
    temperature=0.7,
    api_key=os.getenv("OPENAI_API_KEY")
)
```

1.5 LangGraph Tools

1.5.1 Decorador `@tool`

Define funções como ferramentas para agentes.

```
@tool
def get_server_info(server_id: int) -> str:
    """Obtém informações de um servidor específico."""
    # implementação
    return info
```

1.6 Interface Streamlit

1.6.1 Estado de Sessão (st.session_state)

1.6.1.1 `st.session_state.app`

Grafo compilado do LangGraph.

1.6.1.2 `st.session_state.thread_id`

Identificador do thread: `"streamlit_thread_folha"`

1.6.1.3 `st.session_state.chat_history`

Lista de mensagens (BaseMessage) do histórico completo.

Mensagem Inicial:

```
AIMessage(content="Olá! Sou seu assistente de Folha de Pagamento...")
```

1.6.1.4 `st.session_state.processing_lock`

Bloqueio (bool) para evitar processamento simultâneo.

1.6.2 Componentes Principais

1.6.2.1 Configuração de Página

```
st.set_page_config(
    page_title="Conversa com a Folha",
    page_icon=":100:",
    layout="wide"
)
```

1.6.2.2 Sidebar - Inputs de API

```
groq_api_key = st.sidebar.text_input(
    "🔑 Groq API Key",
    type="password"
)

openai_api_key = st.sidebar.text_input(
    "🔑 OpenAI API Key",
    type="password"
)
```

1.6.2.3 Container de Chat

```
container_chat = st.container(height=500)

with container_chat:
    for msg in st.session_state.chat_history:
        # Renderiza mensagens com avatars
        # 🦙 Groq (Llama3)
        # 🤔 OpenAI (GPT)
        # 🛠️ Ferramenta
        # 👩‍🦰 Usuário
```

1.6.2.4 Input de Chat

```
if prompt := st.chat_input("Faça uma pergunta sobre a Folha de Pagamento ..."):
    st.session_state.chat_history.append(HumanMessage(content=prompt))
    st.rerun()
```

1.6.2.5 Processamento de Mensagens

```
if st.session_state.chat_history and isinstance(
    st.session_state.chat_history[-1], HumanMessage
```

```
):
    with st.spinner("🔄 Consultando Folha de Pagamento e pensando..."):
        current_state = {"messages": st.session_state.chat_history}
        final_state = st.session_state.app.invoke(current_state)

        # Adiciona novas mensagens ao histórico
        new_messages = final_state["messages"][len(current_state["messages"]):]
        st.session_state.chat_history.extend(new_messages)
        st.rerun()
```

1.6.3 Avatares e Identificação de Agentes

Lógica de Identificação:

```
# Verifica menções explícitas
is_groq_explicit = "@groq" in msg.content.lower()
is_openai_explicit = "@openai" in msg.content.lower()

# Conta mensagens AI para alternância
ai_message_index = sum(
    1 for m in st.session_state.chat_history[:i]
    if isinstance(m, AIMessage)
)

# Define avatar
if is_groq_explicit or (ai_message_index % 2 == 0):
    avatar_icon = "🦙"  # Groq/Llama3
    sender_name = "Groq (Llama3)"
elif is_openai_explicit or (ai_message_index % 2 != 0):
    avatar_icon = "🧐"  # OpenAI/GPT
    sender_name = "OpenAI (GPT)"
```

1.7 Variáveis de Ambiente

```
OPENAI_API_KEY          # Chave API OpenAI
GROQ_API_KEY            # Chave API Groq
AGENTICOPS_API_KEY      # Chave API AgenticOps (opcional)
DATABASE_PATH           # Caminho do bank de dados
PYTHONPATH              # Caminho Python (VSCode)
```

1.8 Tipos de Dados

1.8.1 BaseMessage

Classe-base para mensagens no LangChain.

Tipos: - `HumanMessage`: Mensagem do usuário - `AIMessage`: Resposta do agente - `ToolMessage`: Resposta de uma ferramenta - `SystemMessage`: Mensagem do sistema

---

1.9 Configuração AgenticOps

Ver config/agenticops_config.yaml

Recurso principal:

```
from agenticops import AgenticOps

ops = AgenticOps(
    api_key=os.getenv("AGENTICOPS_API_KEY"),
    project_name="FACIN_IA"
)
```

1.10 Tratamento de Erros

1.10.1 Exceções Personalizadas

```
class DatabaseError(Exception):
    """Erro ao acessar banco de dados"""
    pass

class AgentError(Exception):
    """Erro ao executar agente"""
    pass

class ToolError(Exception):
    """Erro ao executar ferramenta"""
    pass
```

1.11 Logging

```
import logging

logger = logging.getLogger(__name__)

logger.info("Mensagem informativa")
logger.warning("Aviso")
logger.error("Erro")
logger.debug("Debug")
```

1.12 Testes

1.12.1 Estrutura de Testes

```
import pytest
from app import execute_database_query

def test_database_query():
    result = execute_database_query("SELECT COUNT(*) FROM servidores")
    assert isinstance(result, str)
    assert len(result) > 0
```

1.13 CI/CD Integration

1.13.1 Validação de Especificação

Arquivo de especificação obrigatório em JSON:

```
{
    "version": "1.0.0",
    "name": "FACIN_IA",
    "description": "Sistema Inteligente Multi-Agentes",
    "modules": [
        {
            "name": "app.py",
            "type": "main",
            "dependencies": ["streamlit", "langchain", "langgraph"]
        }
    ]
}
```

1.14 Recursos Adicionais

- Documentação LangChain
- Documentação LangGraph
- Documentação Streamlit
- AgenticOps Docs

Versão: 1.0.0
Nível de Maturidade: 1