Documentação Técnica - Sistema Memo

Sumário

Introdução

O **Memo** é um sistema web desenvolvido em Python com Flask que permite aos usuários criar, gerenciar e compartilhar memórias de eventos, viagens e festas. O sistema implementa uma arquitetura MVC (Model-View-Controller) completa, com três CRUDs principais: Usuários, Eventos e Fotos.

Tecnologias Utilizadas

- **Backend**: Python 3.x com Flask
- **Frontend**: HTML5, CSS3 (JavaScript mínimo - menos de 30%)
- **Banco de Dados**: SQLite
- **Arquitetura**: MVC (Model-View-Controller)

Arquitetura do Sistema

O sistema segue o padrão arquitetural MVC, separando responsabilidades em três camadas principais:

Model (Modelos)

Responsáveis pela interação com o banco de dados e lógica de negócio relacionada aos dados.

Localização: `/models/`

- `user.py` - Gerencia operações de usuários
- `event.py` - Gerencia operações de eventos
- `photo.py` - Gerencia operações de fotos

View (Visualizações)

Templates HTML que renderizam a interface do usuário.

Localização: `/views/`

- Templates baseados em Jinja2
- Herança de templates através de `base.html`
- Separação de lógica de apresentação

Controller (Controladores)

Processam requisições HTTP, validam dados e coordenam entre Models e Views.

Localização: `/controllers/`

- `auth_controller.py` - Autenticação e autorização
- `event_controller.py` - Gerenciamento de eventos
- `photo_controller.py` - Gerenciamento de fotos

Estrutura de Pastas

```
/memo /models # Modelos de dados (User, Event, Photo) /controllers # Lógica de controle e validação /views # Templates HTML /static # Arquivos estáticos (CSS, imagens) /css # Estilos CSS /img # Imagens estáticas /uploads # Fotos enviadas pelos usuários app.py # Aplicação Flask principal database.py # Configuração do banco de dados run.py # Script de inicialização
```

Banco de Dados

Tabela: users

Armazena informações dos usuários do sistema.

Campos:
- `id` (INTEGER PRIMARY KEY) - Identificador único
- `name` (TEXT NOT NULL) - Nome completo do usuário
- `email` (TEXT NOT NULL UNIQUE) - Email único do usuário
- `password_hash` (TEXT NOT NULL) - Senha criptografada
- `created_at` (TIMESTAMP) - Data de criação

Função Essencial: `init_db()` em `database.py`

```
def init_db(): """Cria todas as tabelas do banco de dados""" # Cria tabela users com constraints e índices
```

Tabela: events

Armazena os eventos criados pelos usuários.

Campos:
- `id` (INTEGER PRIMARY KEY) - Identificador único
- `user_id` (INTEGER NOT NULL) - FK para users
- `title` (TEXT NOT NULL) - Título do evento
- `description` (TEXT NOT NULL) - Descrição detalhada
- `location` (TEXT NOT NULL) - Local do evento
- `date` (DATE NOT NULL) - Data do evento
- `visibility` (TEXT NOT NULL) - 'public' ou 'private'
- `cover_image` (TEXT) - Nome do arquivo da imagem de capa
- `created_at` (TIMESTAMP) - Data de criação

Relacionamento: Um usuário pode ter muitos eventos (1:N)

Tabela: photos

Armazena as fotos vinculadas aos eventos.

Campos:
- `id` (INTEGER PRIMARY KEY) - Identificador único
- `event_id` (INTEGER NOT NULL) - FK para events
- `filename` (TEXT NOT NULL) - Nome do arquivo da foto
- `uploaded_at` (TIMESTAMP) - Data de upload

Relacionamento: Um evento pode ter muitas fotos (1:N)

Índices Criados (para otimização):

- `idx_events_user_id` - Busca rápida de eventos por usuário
- `idx_events_visibility` - Filtro de eventos públicos/privados
- `idx_photos_event_id` - Busca rápida de fotos por evento
- `idx_users_email` - Busca rápida de usuário por email

Páginas do Sistema

1. Página Inicial (Home) - `/`

Arquivo: `views/home.html`

Função no app.py: `home()`

Descrição:
Página pública que apresenta o sistema Memo, suas funcionalidades principais e convida usuários a se cadastrarem ou explorarem eventos públicos.

Elementos Principais:
- Hero section com título e descrição
- Seção de features (Upload, Privacidade, Organização, Compartilhamento)
- Call-to-action para cadastro ou acesso ao dashboard

Fluxo:
```
@app.route('/') def home(): """Renderiza a página inicial""" return
render_template('home.html')
```

2. Explorar Eventos - `/explorar`

Arquivo: `views/explore.html`

Função no app.py: `explore()`

Descrição:
Lista todos os eventos públicos criados por usuários, permitindo que visitantes visualizem memórias compartilhadas.

Função Essencial:
```
@app.route('/explorar') def explore(): """Lista eventos públicos""" events
= Event.find_public_events(limit=20) return
render_template('explore.html', events=events)
```

Modelo Utilizado: `Event.find_public_events()`
- Busca eventos com `visibility = 'public'`
- Ordena por data (mais recentes primeiro)

- Limita a 20 resultados
- Inclui nome do criador através de JOIN

3. Como Funciona - `/como-funciona`

Arquivo: `views/how_it_works.html`

Função no app.py: `how_it_works()`

Descrição:
Página explicativa que detalha o funcionamento do sistema, desde o cadastro até o gerenciamento de eventos e fotos.

Conteúdo:
- Passo a passo de uso do sistema
- Explicação sobre privacidade (público/privado)
- Informações sobre segurança de dados

4. Login - `/login`

Arquivo: `views/login.html`

Função no app.py: `login()`

Descrição:
Página de autenticação que permite usuários fazerem login no sistema.

Fluxo de Funcionamento:

```
@app.route('/login', methods=['GET', 'POST']) def login(): if
request.method == 'POST': email = request.form.get('email', '').strip()
password = request.form.get('password', '') # Validação e autenticação
success, message, user = AuthController.login(email, password) if success
and user: # Cria sessão session['user_id'] = user.id session['user_name'] =
user.name session['user_email'] = user.email return
redirect(url_for('dashboard'))
```

Funções Essenciais:
- `AuthController.login()` - Valida credenciais
- `User.find_by_email()` - Busca usuário no banco
- `User.verify_password()` - Verifica senha hasheada

Segurança:
- Senhas nunca são armazenadas em texto plano
- Hash gerado com `werkzeug.security.generate_password_hash()`
- Verificação com `werkzeug.security.check_password_hash()`

5. Cadastro - `/cadastro`

Arquivo: `views/register.html`

Função no app.py: `register()`

Descrição:
Permite criação de novas contas no sistema.

Validações Realizadas (em `AuthController.register()`):
1. Todos os campos obrigatórios preenchidos
2. Nome com mínimo de 3 caracteres
3. Email em formato válido (regex)
4. Senha com mínimo de 6 caracteres
5. Confirmação de senha igual à senha
6. Email único (não cadastrado anteriormente)

Função Essencial:

```
def register(name, email, password, confirm_password): # Validações if not
AuthController.validate_email(email): return False, "Email inválido.",
None # Verifica se email já existe existing_user =
User.find_by_email(email) if existing_user: return False, "Este email já
está cadastrado.", None # Cria usuário com senha hasheada user =
User.create(name, email, password)
```

6. Redefinir Senha - `/redefinir-senha`

Arquivo: `views/reset_password.html`

Função no app.py: `reset_password()`

Descrição:
Permite que usuários redefinam suas senhas através do email.

Fluxo:
1. Usuário informa email e nova senha
2. Sistema valida email e senha
3. Busca usuário pelo email
4. Atualiza senha com novo hash

Função Essencial:

```
def reset_password(email, new_password, confirm_password): user =
User.find_by_email(email) if user.update_password(new_password): return
True, "Senha redefinida com sucesso!"
```

7. Dashboard - `/dashboard`

Arquivo: `views/dashboard.html`

Função no app.py: `dashboard()`

Proteção: Requer autenticação (`@login_required`)

Descrição:
Painel principal do usuário logado, exibindo todos os seus eventos (públicos e privados).

Função Essencial:

```
@app.route('/dashboard') @login_required def dashboard(): user_id =
session['user_id'] events = Event.find_by_user(user_id) return
render_template('dashboard.html', events=events)
```

Modelo Utilizado: `Event.find_by_user(user_id)`
- Busca todos os eventos do usuário
- Ordena por data (mais recentes primeiro)
- Retorna lista de objetos Event

Funcionalidades:
- Visualizar todos os eventos
- Criar novo evento
- Editar evento existente
- Excluir evento
- Ver detalhes do evento

8. Criar Evento - `/evento/criar`

Arquivo: `views/create_event.html`

Função no app.py: `create_event()`

Proteção: Requer autenticação

Descrição:
Formulário para criação de novos eventos.

Campos do Formulário:
- Título (obrigatório, 3-200 caracteres)
- Descrição (obrigatória, 10-2000 caracteres)
- Local (obrigatório, 3-200 caracteres)
- Data (obrigatória, formato YYYY-MM-DD)
- Visibilidade (público ou privado)

Validações (em `EventController.create()`):

```
def create(user_id, title, description, location, date, visibility): #
Valida título if not EventController.validate_title(title): return False,
"Título deve ter entre 3 e 200 caracteres.", None # Valida descrição if not
EventController.validate_description(description): return False,
"Descrição deve ter entre 10 e 2000 caracteres.", None # Valida data if not
EventController.validate_date(date): return False, "Data inválida. Use o
formato YYYY-MM-DD.", None # Cria evento event = Event.create(user_id,
title, description, location, date, visibility)
```

Fluxo:
1. Usuário preenche formulário
2. Controller valida todos os campos
3. Model cria registro no banco
4. Redireciona para detalhes do evento criado

9. Editar Evento - `/evento//editar`

Arquivo: `views/edit_event.html`

Função no app.py: `edit_event(event_id)`

Proteção: Requer autenticação + verificação de propriedade

Descrição:
Permite edição de eventos existentes.

Verificação de Permissão:
```
event = Event.find_by_id(event_id) if not event.is_owner(user_id):
flash('Você não tem permissão para editar este evento.', 'error') return
redirect(url_for('dashboard'))
```

Função Essencial: `EventController.update()`
- Valida todos os campos
- Verifica permissão do usuário
- Atualiza registro no banco
- Retorna sucesso ou erro

10. Detalhes do Evento - `/evento/`

Arquivo: `views/event_details.html`

Função no app.py: `event_details(event_id)`

Descrição:
Exibe informações completas de um evento, incluindo todas as fotos.

Controle de Acesso:
```
def event_details(event_id): user_id = session.get('user_id') can_view,
event = EventController.can_view(event_id, user_id) if not can_view or not
event: flash('Evento não encontrado ou você não tem permissão.', 'error')
return redirect(url_for('explore'))
```

Lógica de Visibilidade:
- Eventos públicos: visíveis para todos
- Eventos privados: apenas para o dono

Funcionalidades:
- Visualizar informações do evento
- Ver todas as fotos do evento
- Upload de novas fotos (apenas dono)
- Exclusão de fotos (apenas dono)

Funções Essenciais

Autenticação e Autorização

`AuthController.login(email, password)`

Localização: `controllers/auth_controller.py`

Função: Realiza autenticação do usuário.

Fluxo:
1. Valida formato do email
2. Busca usuário pelo email: `User.find_by_email(email)`
3. Verifica senha: `user.verify_password(password)`
4. Retorna tupla (success, message, user)

Código Essencial:

```
def login(email, password): user = User.find_by_email(email) if not user:
return False, "Email ou senha incorretos.", None if not
user.verify_password(password): return False, "Email ou senha
incorretos.", None return True, "Login realizado com sucesso!", user
```

`@login_required` (Decorator)

Localização: `app.py`

Função: Protege rotas que requerem autenticação.

Implementação:

```
def login_required(f): @wraps(f) def decorated_function(*args, **kwargs):
if 'user_id' not in session: flash('Você precisa fazer login.', 'error')
return redirect(url_for('login')) return f(*args, **kwargs) return
decorated_function
```

Uso: Aplicado em rotas protegidas:

```
@app.route('/dashboard') @login_required def dashboard(): # Código da
função
```

Gerenciamento de Eventos

`EventController.create()`

Localização: `controllers/event_controller.py`

Função: Cria um novo evento com validações completas.

Validações Realizadas:
1. Título: 3-200 caracteres

2. Descrição: 10-2000 caracteres
3. Local: 3-200 caracteres
4. Data: formato YYYY-MM-DD válido
5. Visibilidade: 'public' ou 'private'

Fluxo:

```
def create(user_id, title, description, location, date, visibility): # Valida todos os campos if not validate_title(title): return False, "Título inválido.", None # Cria evento no banco event = Event.create(user_id, title, description, location, date, visibility) if event: return True, "Evento criado com sucesso!", event
```

`EventController.can_view(event_id, user_id)`

Localização: `controllers/event_controller.py`

Função: Verifica se usuário pode visualizar um evento.

Lógica:

```
def can_view(event_id, user_id=None): event = Event.find_by_id(event_id) if not event: return False, None # Eventos públicos: qualquer um pode ver if event.visibility == 'public': return True, event # Eventos privados: apenas o dono if user_id and event.is_owner(user_id): return True, event return False, None
```

Gerenciamento de Fotos

`PhotoController.upload()`

Localização: `controllers/photo_controller.py`

Função: Processa upload de fotos para eventos.

Validações:
1. Arquivo enviado existe
2. Extensão permitida (PNG, JPG, JPEG, GIF, WEBP)
3. Evento existe
4. Usuário é dono do evento

Fluxo Completo:

```
def upload(event_id, user_id, file, upload_folder): # 1. Valida arquivo if not allowed_file(file.filename): return False, "Tipo de arquivo não permitido.", None # 2. Verifica permissão event = Event.find_by_id(event_id) if not event.is_owner(user_id): return False, "Sem permissão.", None # 3. Gera nome único unique_filename = f"{uuid.uuid4().hex}.{file_extension}" # 4. Salva arquivo físico file.save(file_path) # 5. Cria registro no banco photo = Photo.create(event_id, unique_filename) return True, "Foto enviada com sucesso!", photo
```

Segurança:
- Nome do arquivo sanitizado com `secure_filename()`
- Nome único gerado com UUID
- Validação de extensão
- Verificação de permissão

`download_file(filename)`

Localização: `app.py`

Função: Serve arquivos de fotos para download/visualização.

Implementação:

```
@app.route('/download/') def download_file(filename): return
send_from_directory( app.config['UPLOAD_FOLDER'], filename,
as_attachment=False # Exibe no navegador )
```

Modelos de Dados

`User.create(name, email, password)`

Localização: `models/user.py`

Função: Cria novo usuário no banco de dados.

Processo:
1. Gera hash da senha: `generate_password_hash(password)`
2. Insere no banco
3. Retorna objeto User criado

Código:

```
@staticmethod def create(name, email, password): password_hash =
generate_password_hash(password) cursor.execute(''' INSERT INTO users
(name, email, password_hash) VALUES (?, ?, ?) ''', (name, email,
password_hash)) conn.commit() user_id = cursor.lastrowid return
User.find_by_id(user_id)
```

`Event.find_public_events(limit=20)`

Localização: `models/event.py`

Função: Busca eventos públicos mais recentes.

Query SQL:

```
cursor.execute(''' SELECT e.*, u.name as user_name FROM events e JOIN users
u ON e.user_id = u.id WHERE e.visibility = 'public' ORDER BY e.date DESC,
e.created_at DESC LIMIT ? ''', (limit,))
```

Otimização: Usa índice `idx_events_visibility` para busca rápida.

Fluxo de Funcionamento

Fluxo de Cadastro e Login

```
1. Usuário acessa /cadastro ↓ 2. Preenche formulário (nome, email, senha) ↓
3. AuthController.register() valida dados ↓ 4. User.create() cria registro
no banco ↓ 5. Redireciona para /login ↓ 6. Usuário faz login ↓ 7.
AuthController.login() valida credenciais ↓ 8. Cria sessão
(session['user_id']) ↓ 9. Redireciona para /dashboard
```

Fluxo de Criação de Evento

```
1. Usuário logado acessa /evento/criar ↓ 2. Preenche formulário (título,
descrição, local, data, visibilidade) ↓ 3. EventController.create() valida
todos os campos ↓ 4. Event.create() insere no banco ↓ 5. Redireciona para
/evento/ (detalhes)
```

Fluxo de Upload de Foto

```
1. Usuário acessa /evento/ ↓ 2. Seleciona arquivo de imagem ↓ 3.
PhotoController.upload() valida: - Tipo de arquivo permitido - Permissão do
usuário ↓ 4. Gera nome único (UUID) ↓ 5. Salva arquivo em /uploads/ ↓ 6.
Photo.create() cria registro no banco ↓ 7. Foto aparece na página do evento
```

Fluxo de Visualização de Evento

```
1. Usuário acessa /evento/ ↓ 2. EventController.can_view() verifica: -
Evento existe? - É público? → Permite acesso - É privado? → Verifica se é
dono ↓ 3. Photo.find_by_event() busca fotos ↓ 4. Renderiza template com
dados
```

Sistema de Autenticação

Sessões

O sistema utiliza sessões do Flask para manter o estado de autenticação.

Criação de Sessão (após login):

```
session['user_id'] = user.id session['user_name'] = user.name
session['user_email'] = user.email
```

Verificação de Sessão:

```
if 'user_id' not in session: # Usuário não autenticado
```

Limpeza de Sessão (logout):

```
session.clear()
```

Hash de Senhas

Geração de Hash:

```
from werkzeug.security import generate_password_hash password_hash =
generate_password_hash(password) # Resultado: pbkdf2:sha256:260000$...
```

Verificação de Hash:

```
from werkzeug.security import check_password_hash if
check_password_hash(user.password_hash, password): # Senha correta
```

Segurança:
- Algoritmo: PBKDF2 com SHA-256
- Iterações: 260.000 (padrão Werkzeug)
- Salt automático e único por senha

Sistema de Upload

Validação de Arquivos

Extensões Permitidas:

```
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'webp'}
```

Função de Validação:

```
def allowed_file(filename): return '.' in filename and \
filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

Sanitização de Nomes

Problema: Nomes de arquivo podem conter caracteres perigosos.

Solução: `secure_filename()` do Werkzeug

```
from werkzeug.utils import secure_filename original_filename =
secure_filename(file.filename) # Remove caracteres especiais, espaços,
etc.
```

Geração de Nomes Únicos

Problema: Múltiplos uploads com mesmo nome causariam conflito.

Solução: UUID hexadecimal

```
import uuid unique_filename = f"{uuid.uuid4().hex}.{file_extension}" #
Exemplo: a3f5b2c1d4e6f7a8b9c0d1e2f3a4b5c6.png
```

Armazenamento

Estrutura:

```
/uploads/ a3f5b2c1d4e6f7a8b9c0d1e2f3a4b5c6.png
92c3297526cb4f52a26a4a6c85dd8f13.jpg ...
```

Banco de Dados:
- Armazena apenas o nome do arquivo
- Caminho completo construído dinamicamente
- Relacionamento através de `event_id`

Validações Implementadas

Validação de Email

Regex Pattern:

```
pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$' return
bool(re.match(pattern, email))
```

Validação de Data

Formato Esperado: YYYY-MM-DD

Validação:

```
try: datetime.strptime(date_str, '%Y-%m-%d') return True except
ValueError: return False
```

Validação de Campos Obrigatórios

Implementação:

```
if not title or not description or not location or not date: return False,
"Todos os campos são obrigatórios.", None
```

Aplicado em:
- Cadastro de usuário
- Criação de evento
- Edição de evento
- Upload de foto

Segurança

Proteção de Rotas

Decorator `@login_required`:
- Verifica se usuário está autenticado
- Redireciona para login se não estiver
- Aplicado em todas as rotas protegidas

Verificação de Permissões

Propriedade de Evento:
```
def is_owner(self, user_id): return self.user_id == user_id
```

Aplicado em:
- Edição de evento
- Exclusão de evento
- Upload de foto
- Exclusão de foto

Limite de Tamanho de Arquivo

Configuração:
```
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024 # 16MB
```

SQL Injection Prevention

Uso de Parâmetros:
```
cursor.execute('SELECT * FROM users WHERE email = ?', (email,)) # Nunca:
cursor.execute(f'SELECT * FROM users WHERE email = {email}')
```

Conclusão

O sistema Memo implementa uma arquitetura MVC completa e robusta, com separação clara de responsabilidades entre Models, Views e Controllers. Todas as funcionalidades essenciais foram implementadas com validações adequadas, controle de acesso e segurança.

Principais Características

1. **Arquitetura MVC**: Separação clara de responsabilidades
2. **Segurança**: Senhas hasheadas, sessões, validações
3. **Validações Completas**: Todos os campos validados
4. **Controle de Acesso**: Eventos públicos/privados
5. **Upload Seguro**: Validação de tipo, sanitização, nomes únicos
6. **Interface Moderna**: CSS responsivo e design limpo
7. **Banco Otimizado**: Índices para queries rápidas

Funcionalidades Implementadas

■ CRUD completo de Usuários
■ CRUD completo de Eventos
■ CRUD completo de Fotos
■ Sistema de autenticação com sessões
■ Controle de privacidade (público/privado)
■ Upload e download de fotos
■ Validações em todos os formulários
■ Interface responsiva e moderna

O sistema está pronto para uso e pode ser facilmente expandido com novas funcionalidades seguindo os mesmos padrões arquiteturais estabelecidos.

Documentação gerada em: 2024
Versão do Sistema: 1.0
Autor: Sistema Memo