



# DESENVOLVIMENTO WEB



## SIMULADOR DE VESTIBULAR ONLINE



Integração com o Banco de Dados

### Link do Video

[https://drive.google.com/file/d/1k0VsR290acD7HBkT5Zmk4XktncRPjMUp/view?  
usp=sharing](https://drive.google.com/file/d/1k0VsR290acD7HBkT5Zmk4XktncRPjMUp/view?usp=sharing)

**Autores:**

Aluno 1: Julio Cesar Farias

Aluno 2: Michael Pariz Pereira

**Professor:**

*Luiz Antonio  
Lima Rodrigues*

# Objetivo geral

- Objetivo Geral: Desenvolver uma aplicação Web que utiliza Inteligência Artificial para gerar provas sob demanda.

# Objetivos específicos:

- Integrar API externa (Google Gemini) para criação de conteúdo.
  - Implementar persistência de dados (Histórico de Simulados e Usuários).
  - Garantir segurança via Autenticação JWT e Hash de senhas.





# Modelagem do Banco de Dados (ORM)

- O usuário define as opções do simulado no componente “SelecaoSimulado”
- A página detecta que a API deve ser usada
- Ela chama a função apiPost



```
# Tabela de Usuários
class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    username = Column(String, unique=True, index=True)
    hashed_password = Column(String)
    role = Column(String, default="free")

    # Relacionamento: Um usuário tem vários simulados
    simulados = relationship("SimuladoDB", back_populates="dono")

    # Chave Estrangeira (Liga o simulado ao usuário que criou)
    user_id = Column(Integer, ForeignKey("users.id"))
    dono = relationship("User", back_populates="simulados")

    # Relacionamento com as questões
    questoes = relationship("QuestaoDB", back_populates="simulado", cascade="all, delete-orphan")

# NOVA TABELA: Detalhes das Questões
class QuestaoDB(Base):
    __tablename__ = "questoes_salvas"

    id = Column(Integer, primary_key=True, index=True)

    # Liga a questão ao Simulado pai
    simulado_id = Column(Integer, ForeignKey("historico_simulados.id"))
    simulado = relationship("SimuladoDB", back_populates="questoes")

    enunciado = Column(String)
    # Vamos salvar as opções como JSON (lista de textos) para facilitar
    opcoes = Column(JSON)
    resposta_correta_idx = Column(Integer)
```



# Integração: Persistência Transacional (Gerar Simulado)

função orquestrada para chamada da IA e realizar uma transação complexa no banco, salvando primeiro o simulado para obter o ID e, em seguida, iterando para salvar cada questão vinculada.



```
# --- O Endpoint Principal ---  
  
@app.post("/gerar-simulado/{vestibular_id}", response_model=list[QuestaoResponse])  
async def gerar_simulado(  
    vestibular_id: str,  
    request_data: SimuladoRequest,  
    current_user: models.User = Depends(get_current_user), # Recuperamos a segurança  
    db: Session = Depends(get_db)  
):  
  
    # 5. Cria o Cabeçalho do Simulado  
    novo_simulado = models.SimuladoDB(  
        vestibular=vestibular_id,  
        materia=request_data.materia,  
        dificuldade=request_data.dificuldade,  
        num_questoes=request_data.numQuestoes,  
        user_id=current_user.id # Salva o ID do usuário logado  
    )  
    db.add(novo_simulado)  
    db.commit()  
    db.refresh(novo_simulado) # Pega o ID gerado (ex: 1, 2, 3...)  
  
    print(f"-- [DEBUG] 6. Cabeçalho salvo (ID: {novo_simulado.id}). Salvando as questões... --")  
  
    # 6. Salva cada questão na tabela 'questoes_salvas'  
    questoes_formatadas = []  
    for q in questoes_json:  
        # Prepara o objeto para devolver pro Front-end (QuestaoResponse)  
        q['vestibular'] = vestibular_id.upper()  
        q['materia'] = request_data.materia  
        q['dificuldade'] = request_data.dificuldade  
        questoes_formatadas.append(q)  
  
        # Salva no Banco (QuestaoDB)  
        # Nota: q['opcoes'] vem da IA como [{id:0, texto:"A"}, ...].  
        # Aqui extraímos só o texto para salvar no banco.  
        lista_opcoes_texto = [opt['texto'] for opt in q['opcoes']]  
  
        nova_questao = models.QuestaoDB(  
            simulado_id=novo_simulado.id, # Liga ao simulado pai  
            enunciado=q['enunciado'],  
            opcoes=lista_opcoes_texto,  
            resposta_correta_idx=q['respostaCorreta']  
        )  
        db.add(nova_questao)  
  
    db.commit() # Confirma o salvamento das questões  
    print("-- [DEBUG] 7. Tudo salvo! Retornando para o Front. --")  
  
    return questoes_formatadas
```



# Integração: Consultas Relacionais (Histórico e Detalhes)

```
@app.get("/historico", response_model=list[SimuladoHistorico])
async def ler_historico(
    current_user: models.User = Depends(get_current_user), # Exige login
    db: Session = Depends(get_db)
):
    """
    Lista APENAS os simulados do usuário logado.
    """
    historico = db.query(models.SimuladoDB) \
        .filter(models.SimuladoDB.user_id == current_user.id) \
        .order_by(models.SimuladoDB.data_criacao.desc()) \
        .all()

    return historico
```

Implementação de consultas seguras que filtram os dados baseados no ID do usuário autenticado (current\_user), garantindo que um aluno não veja o histórico de outro



# INTEGRAÇÃO: SEGURANÇA E AUTENTICAÇÃO NO BANCO

- Integridade: Verificação de duplicidade de usuários no banco.
- Segurança: Armazenamento de senhas criptografadas (Hash).
- Acesso: Geração de Token JWT para autenticação contínua.



```
# --- Endpoint de Registro (AGORA SALVA NO DB) ---
@app.post("/users/", response_model=UserResponse)
async def create_user(user: UserCreate, db: Session = Depends(get_db)):
    """Registra um novo usuário no BANCO DE DADOS."""

    # 1. Verifica se já existe usando SUA função
    db_user = get_user_by_username(db, username=user.username)

    if db_user:
        raise HTTPException(status_code=400, detail="Usuário (username) já registrado")

    # 2. Criptografa a senha
    hashed_password = get_password_hash(user.password)

    # 3. Cria o objeto do Banco (models.User)
    new_user = models.User(
        username=user.username,
        hashed_password=hashed_password,
        role=user.role
    )

    # 4. Salva no SQLite
    db.add(new_user)
    db.commit()
    db.refresh(new_user)

    return new_user

# --- Endpoint de Login (AGORA LÊ DO DB) ---
@app.post("/token", response_model=Token)
async def login_for_access_token(
    form_data: Annotated[OAuth2PasswordRequestForm, Depends()],
    db: Session = Depends(get_db) # <--- Injetamos o banco
):
    # --- MUDANÇA AQUI: Usamos a SUA função 'get_user_by_username' ---
    user = get_user_by_username(db, username=form_data.username)
    # ----

    # Verifica se existe e se a senha bate
    if not user or not verify_password(form_data.password, user.hashed_password):
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Usuário ou senha incorretos",
            headers={"WWW-Authenticate": "Bearer"}
        )

    # Cria o token
    access_token_expires = timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": user.username, "role": user.role},
        expires_delta=access_token_expires
    )

    return {"access_token": access_token, "token_type": "bearer"}
```



# INTEGRAÇÃO: SEGURANÇA E AUTENTICAÇÃO NO BANCO

```
# --- O Endpoint Principal ---\n\n@app.post("/gerar-simulado/{vestibular_id}", response_model=list[QuestaoResponse])\nasync def gerar_simulado(\n    vestibular_id: str,\n    request_data: SimuladoRequest,\n    current_user: models.User = Depends(get_current_user), # Recuperamos a segurança\n    db: Session = Depends(get_db)\n):
```

