Para o seu cenário de **controle financeiro jurídico**, a escolha ideal para o backend é o **FastAPI**.

Embora o Django seja excelente para sistemas administrativos padrão ("CRUDs" puros), o **FastAPI** leva vantagem no seu projeto por três motivos críticos:

1. **Performance em Automações:** O processamento assíncrono do FastAPI é superior para lidar com as rotinas de importação de OFX e chamadas de APIs bancárias sem travar o sistema.
2. **Validação de Dados Estrita:** Com o uso do *Pydantic*, o FastAPI garante que os dados financeiros (como o `valor_causa` e `percentual_exito`) cheguem ao banco com o tipo e a precisão exatos, reduzindo erros de arredondamento.
3. **Documentação Automática:** Como você terá um frontend em React consumindo essa API, o FastAPI gera o *Swagger* (documentação) instantaneamente, facilitando muito o desenvolvimento.

---

## 1. Arquitetura da Stack Sugerida

- **Linguagem:** Python 3.10+
- **Framework Web:** FastAPI
- **ORM (Acesso ao Banco):** SQLAlchemy ou SQLModel (perfeito para integração com Pydantic)
- **Banco de Dados:** PostgreSQL (com suporte a campos `NUMERIC` para precisão financeira)
- **Frontend:** React com Tailwind CSS (para uma UI limpa) e TanStack Query (para sincronização de dados)

---

## 2. Estrutura de Pastas do Backend

Para manter o projeto organizado e escalável:

```
/backend
├── app
│   ├── api             # Endpoints (Routes)
│   ├── core            # Configurações (Segurança, Variáveis de Ambiente)
│   ├── models          # Definição das tabelas (SQLAlchemy)
│   ├── schemas         # Validação de dados (Pydantic)
│   ├── services        # Lógica de negócio (Cálculo de Êxito, Regras de OFX)
│   └── workers         # Tarefas agendadas (Geração de mensalidades)
├── migrations          # Histórico do banco (Alembic)
└── main.py             # Ponto de entrada
```

---

## 3. Implementação do "Coração" do Sistema (Exemplo)

Veja como ficaria a lógica de **Cálculo de Êxito Automático** usando a elegância do Python no FastAPI:

```
# app/services/financeiro.py

from decimal import Decimal
```

```python
def calcular_previsao_exito(valor_causa: Decimal, percentual: Decimal) -> Decimal:
    """
    Calcula o valor previsto de honorários.
    Garante que a precisão financeira seja mantida.
    """
    if percentual < 0 or percentual > 100:
        raise ValueError("Percentual deve estar entre 0 e 100")

    return (valor_causa * (percentual / Decimal("100"))).quantize(Decimal("0.01"))
```

---

Priorizar a segurança é a decisão mais prudente para um sistema jurídico. Dados de faturamento, detalhes de processos e movimentações bancárias exigem um nível de proteção rigoroso. No **FastAPI**, implementaremos isso usando o padrão **OAuth2 com JWT (JSON Web Tokens)** e uma camada adicional de **2FA (Autenticação de Dois Fatores)**.

Projeto lógico da segurança:

---

## 1. O Fluxo de Autenticação

Para garantir que apenas pessoas autorizadas acessem os dados, o fluxo seguirá este caminho:

1. **Identificação:** Usuário insere e-mail e senha.
2. **Validação Primária:** O sistema verifica o hash da senha no banco de dados.
3. **Desafio 2FA:** Se a senha estiver correta, o sistema **não** loga o usuário ainda. Ele solicita um código de 6 dígitos.
4. **Verificação:** O usuário fornece o código (gerado via app como Google Authenticator ou enviado por e-mail).
5. **Emissão de Token:** Somente após o 2FA, o sistema gera um **Access Token (JWT)** de curta duração e um **Refresh Token** de longa duração.

---

## 2. Estrutura da Tabela de Usuários

Precisamos adaptar a tabela de usuários para suportar o 2FA:

- `id`: UUID
- `email`: VARCHAR (Unique)
- `password_hash`: VARCHAR (Usando algoritmo **Argon2** ou **Bcrypt**)
- `secret_2fa`: VARCHAR (Chave secreta para gerar o QR Code do Google Authenticator)
- `is_2fa_enabled`: BOOLEAN (Default: `false`)
- `role`: ENUM ('ADMIN', 'ANALISTA', 'ADVOGADO') - *Para controle de permissões.*

---

## 3. Implementação do 2FA com Python

No FastAPI, utilizaremos a biblioteca `pyotp` para gerenciar as senhas temporárias (TOTP).

- **Registro:** O sistema gera uma chave secreta e exibe um **QR Code** para o usuário escanear.
- **Login:** O backend recebe o código enviado pelo frontend e valida:

```
import pyotp

# Exemplo de verificação
totp = pyotp.TOTP(user.secret_2fa)
if totp.verify(codigo_digitado):
    # Gera o JWT definitivo
    return generate_access_token(user.id)
```

---

## 4. Controle de Acesso por Nível (RBAC)

Nem todo mundo deve ver tudo. No projeto lógico, definiremos permissões:

| Nível | Permissões |
|---|---|
| **ADMIN** | Acesso total, gestão de usuários e configurações bancárias. |
| **ANALISTA** | Operação financeira, importação de OFX e conciliação. |
| **ADVOGADO** | Visualização de honorários de seus processos e dashboard de êxito. |

---

## 5. Medidas de Segurança Adicionais

Como peer de desenvolvimento, recomendo implementarmos também:

1. **Rate Limiting:** Bloquear temporariamente IPs que tentarem errar a senha ou o 2FA múltiplas vezes (Proteção contra força bruta).
2. **HTTPS Obrigatório:** Criptografia em trânsito para que ninguém intercepte os dados bancários na rede.
3. **Audit Log:** Uma tabela para registrar quem visualizou ou alterou dados sensíveis (ex: "*Usuário X alterou o valor de êxito do Processo Y*").

---

O design de interface (**UI**) e a experiência do usuário (**UX**) são fundamentais em sistemas financeiros: se a tela for confusa, o analista cometerá erros; se o 2FA for burocrático demais, as pessoas tentarão desativá-lo.

Vamos detalhar a jornada do usuário no acesso ao sistema, focando em **limpeza visual** e **segurança sem atrito**.

---

## 1. Fluxo de Login em Dois Passos (Split Login)

Em vez de pedir tudo em uma única tela, dividimos o processo para reduzir a carga cognitiva.

### Passo 1: Identificação

- **Visual:** Uma tela minimalista com o logo da consultoria jurídica à esquerda e o formulário à direita.

- **Campos:** E-mail e Senha.
- **UX:** Botão "Entrar" que, após validado, faz uma transição suave (slide) para a tela de 2FA.

### Passo 2: Verificação 2FA

- **Visual:** Seis campos de entrada individuais (um para cada dígito do código).
- **UX:** * Foco automático no primeiro campo.
- Ao digitar um número, o cursor pula automaticamente para o próximo.
- Opção "Lembrar deste dispositivo por 30 dias" (isso armazena um *fingerprint* seguro no navegador, evitando o 2FA diário em máquinas confiáveis).

---

## 2. Tela de Configuração Inicial (Setup 2FA)

Na primeira vez que o usuário logar (ou quando o Admin exigir a ativação), ele verá uma tela de "Proteja sua Conta":

1. **Instrução:** "Baixe o Google Authenticator ou Authy".
2. **Ação:** Exibição do **QR Code** gerado pelo Python (`pyotp`).
3. **Backup:** Exibição de 5 **Códigos de Recuperação** (para caso o usuário perca o celular).

- *Importante:* O sistema só deixa ele prosseguir após ele marcar um checkbox: "Eu salvei meus códigos de recuperação em um local seguro".

---

## 3. Dashboard do Analista: A "Home" Financeira

Ao entrar, o analista não deve ver apenas tabelas. Ele deve ver o **estado de saúde** da empresa através de um layout de "Cards e Gráficos".

### Layout Sugerido:

- **Topo (KPIs):**
- Card 1: Saldo Disponível (Bancos + Caixas).
- Card 2: A Pagar Hoje (Contas com vencimento para a data atual).
- Card 3: Fatura Cartão (Valor acumulado até o momento).
- **Centro (Ação):**
- Área de "Drag & Drop" para **Importação de OFX**. O analista arrasta o arquivo e o sistema já abre a prévia da conciliação.
- **Lateral (Sidebar):**
- Menu vertical: Dashboard, Processos, Lançamentos, Cartões, Relatórios e Configurações.

---

## 4. Design das Listagens (Contas a Receber/Pagar)

Como o jurídico tem muitos dados, a tabela de lançamentos deve ser uma "DataGrid" poderosa:

- **Status Colorido:**
- 🟢 **Verde:** Pago.
- 🟡 **Amarelo:** Pendente.
- 🔵 **Azul:** Previsão de Êxito (Sem data).
- 🔴 **Vermelho:** Atrasado.
- **Filtros Rápidos:** Botões no topo da tabela para filtrar por "Somente Êxito", "Somente Cartão" ou "Somente Recorrentes".

---

## 5. Micro-interações de Segurança

Para manter o analista alerta:

- **Timeout de Sessão:** Após 15 minutos de inatividade, o sistema escurece a tela e pede apenas o código 2FA ou a senha para reativar (sem deslogar completamente).
- **Feedback de Erro:** Se o 2FA estiver errado, o campo vibra levemente em vermelho.

---

## Modelagem de Dados Completa

Vamos traduzir todo esse projeto lógico para a estrutura de classes do **SQLAlchemy**. Esta etapa é crucial porque define como o **FastAPI** conversará com o **PostgreSQL**.

Como estamos usando Python, utilizaremos a tipagem do `SQLModel` (que une SQLAlchemy e Pydantic) ou o SQLAlchemy puro com `Mapped`. Vou seguir com a sintaxe moderna do SQLAlchemy 2.0 por ser o padrão de mercado para alta performance.

---

## 1. O Mapa do Banco de Dados (Models)

### A. Segurança e Usuários

```python
from sqlalchemy.orm import DeclarativeBase, Mapped, mapped_column, relationship
from sqlalchemy import String, Decimal, ForeignKey, Date, Enum, Boolean
from typing import List, Optional
import uuid

class Base(DeclarativeBase):
    pass

class Usuario(Base):
    __tablename__ = "usuarios"

    id: Mapped[uuid.UUID] = mapped_column(primary_key=True, default=uuid.uuid4)
    email: Mapped[str] = mapped_column(String(255), unique=True, nullable=False)
```

```
    password_hash: Mapped[str] = mapped_column(String(255), nullable=False)
    secret_2fa: Mapped[Optional[str]] = mapped_column(String(64))
    is_2fa_enabled: Mapped[bool] = mapped_column(Boolean, default=False)
    role: Mapped[str] = mapped_column(String(20), default="ANALISTA") # ADMIN,
ANALISTA, ADVOGADO
```

### B. O Core Jurídico (Processos e Honorários)

Aqui inserimos a lógica de percentual de êxito que discutimos.

```
class Processo(Base):
    __tablename__ = "processos"

    id: Mapped[uuid.UUID] = mapped_column(primary_key=True, default=uuid.uuid4)
    numero_cnj: Mapped[str] = mapped_column(String(30), unique=True)
    titulo: Mapped[str] = mapped_column(String(255))
    percentual_exito: Mapped[Decimal] = mapped_column(Decimal(5, 2))
    id_cliente: Mapped[uuid.UUID] = mapped_column(ForeignKey("participantes.id"))

    # Relacionamentos
    lancamentos: Mapped[List["Lancamento"]] =
relationship(back_populates="processo")
```

### C. O Motor Financeiro (Lançamentos e Cartões)

Esta tabela é a mais flexível, permitindo datas nulas para os êxitos.

```
class Lancamento(Base):
    __tablename__ = "lancamentos"

    id: Mapped[uuid.UUID] = mapped_column(primary_key=True, default=uuid.uuid4)
    descricao: Mapped[str] = mapped_column(String(255))
    valor_previsto: Mapped[Decimal] = mapped_column(Decimal(15, 2))
    valor_realizado: Mapped[Optional[Decimal]] = mapped_column(Decimal(15, 2))

    data_vencimento: Mapped[Optional[Date]] = mapped_column(Date) # Null para
êxito sem prazo
    data_pagamento: Mapped[Optional[Date]] = mapped_column(Date)

    status: Mapped[str] = mapped_column(String(20)) # PAGO, PENDENTE, PREVISAO,
ATRASADO
    natureza: Mapped[str] = mapped_column(String(20)) # EXITO, FIXO, CARTAO,
REEMBOLSO

    # Chaves Estrangeiras
    id_processo: Mapped[Optional[uuid.UUID]] =
mapped_column(ForeignKey("processos.id"))
```

```
    id_cartao: Mapped[Optional[uuid.UUID]] =
mapped_column(ForeignKey("cartoes_credito.id"))
    id_centro_custo: Mapped[int] = mapped_column(ForeignKey("centros_custo.id"))

    processo: Mapped["Processo"] = relationship(back_populates="lancamentos")
```

---

## 2. A Tabela de Cartão de Crédito

Como você mencionou o uso frequente de cartões para softwares, esta tabela organiza os limites e vencimentos.

```
class CartaoCredito(Base):
    __tablename__ = "cartoes_credito"

    id: Mapped[uuid.UUID] = mapped_column(primary_key=True, default=uuid.uuid4)
    nome: Mapped[str] = mapped_column(String(50)) # Ex: Visa Escritório
    dia_fechamento: Mapped[int] = mapped_column()
    dia_vencimento: Mapped[int] = mapped_column()
    limite: Mapped[Decimal] = mapped_column(Decimal(15, 2))
```

---

## 3. Por que essa modelagem é eficiente?

1. **Precisão Decimal:** Usamos `Decimal(15, 2)` em vez de `Float` para evitar que R$ 1.000,00 vire R$ 999,99999 após um cálculo de porcentagem.
2. **Integridade Referencial:** Se você tentar deletar um cliente que possui um processo com honorários pendentes, o banco de dados impedirá, protegendo o histórico financeiro.
3. **UUIDs:** Usamos UUIDs em vez de IDs sequenciais (1, 2, 3...) para aumentar a segurança, dificultando que alguém tente "adivinhar" a URL de um processo ou lançamento.

---

## 4. O Próximo Passo: Service Layer (A Inteligência)

Com o banco de dados desenhado em código, o próximo passo é criar os **Services**. Os Services são funções Python que executam as regras de negócio que definimos, como:

- `importar_ofx(arquivo)`: Onde o algoritmo de matching que discutimos vai morar.
- `gerar_previsao_exito(id_processo)`: Onde o sistema calcula o valor baseado no percentual e cria o lançamento.

**Gostaria que eu detalhasse agora o código do Service que processa o arquivo OFX e faz o "Matching" com esses modelos que acabamos de criar?**

Próximo Passo

Com o modelo de segurança definido, o que você prefere fazer agora?

1. **Esquematizar os Endpoints de Login:** Definir as rotas `/auth/login`, `/auth/verify-2fa` e `/auth/setup-2fa`.
2. **Modelagem de Dados Completa:** Traduzir todas as tabelas (Usuários, Processos, Lançamentos) para classes do SQLAlchemy.
3. **Definição do Layout de Login:** Pensar na experiência do usuário (UX) para que o 2FA não seja um fardo, mas uma proteção simples.

**Qual desses caminhos você quer detalhar agora?**