🎯 EXERCÍCIO 1: AGENTE DOCUMENTADOR DOS ENDPOINTS

Nível: Intermediário | Tempo: 20 minutos

Contexto

Você tem os endpoints da API no `/backend/app/api/v1/` mas falta documentação detalhada. Use um agente para criar documentação OpenAPI completa.

Tarefa

```
@workspace você é meu API Documentation Agent para o FinanceFlow.

Analise os arquivos em backend/app/api/v1/:
- auth.py
- users.py
- accounts.py
- transactions.py
- categories.py
- budgets.py

Para CADA endpoint:
1. Extraia a rota, método HTTP e parâmetros
2. Crie documentação OpenAPI incluindo:
   - Summary e description
   - Parameters (path, query, body)
   - Request body schema
   - Response schemas (200, 400, 401, 404)
   - Security requirements (JWT)
   - Examples de request/response

3. Gere arquivo docs/api/openapi.yaml completo
4. Crie também um README.md em docs/api/ com:
   - Tabela resumo de todos endpoints
   - Fluxo de autenticação
   - Exemplos de uso com curl
   - Rate limiting e quotas

Formato final: OpenAPI 3.0 válido
```

Entregáveis

- [ ] `docs/api/openapi.yaml` completo
- [ ] `docs/api/README.md` com exemplos práticos
- [ ] Documentação de todos os 30+ endpoints
- [ ] Schemas de request/response definidos

Validação

```
# Validar OpenAPI gerado
npx @apidevtools/swagger-cli validate docs/api/openapi.yaml
```

🎯 EXERCÍCIO 2: AGENTE ARQUITETO PARA SERVICES LAYER

Nível: Avançado | Tempo: 30 minutos

Contexto

O backend tem lógica misturada nos endpoints. Precisa criar uma camada de serviços adequada em `/backend/app/services/`.

Estrutura Atual

```
 backend/app/api/v1/transactions.py - lógica nos endpoints
backend/app/models/ - modelos SQLAlchemy
backend/app/schemas/ - validação Pydantic
backend/app/services/ - pasta vazia
```

Comando do Agente

```
@workspace você é meu Service Layer Architect para FinanceFlow.

ANÁLISE:
1. Examine backend/app/api/v1/transactions.py
2. Identifique toda lógica de negócios nos endpoints
3. Liste operações que devem ir para service layer

REFATORAÇÃO:
1. Crie backend/app/services/transaction_service.py com:
   - Classe TransactionService
   - Métodos para cada operação de negócio:
     * create_transaction()
     * update_transaction()
     * delete_transaction()
     * get_user_transactions()
     * get_transaction_by_category()
     * calculate_monthly_summary()
     * apply_recurring_transactions()

2. Cada método deve:
   - Receber DTOs/Schemas como parâmetros
   - Fazer validações de negócio
   - Chamar repositories/DAOs
   - Retornar objetos tipados
   - Ter error handling apropriado

3. Refatore transactions.py para:
   - Apenas receber requests
   - Chamar service
   - Retornar responses
   - Zero lógica de negócio

4. Crie também:
   - services/base_service.py - classe base
   - services/__init__.py - exports
   - tests/services/test_transaction_service.py

Mantenha 100% de compatibilidade com frontend.
```

Entregáveis

- ☐ Service layer completa implementada
- ☐ Endpoints refatorados sem lógica
- ☐ Testes unitários dos services
- ☐ Documentação da arquitetura em `/docs/architecture/`

EXERCÍCIO 3: AGENTE DE TESTES PARA MODELS

Nível: Intermediário | Tempo: 25 minutos

Contexto

Os models em `/backend/app/models/` não têm testes. Criar suite completa com fixtures realistas.

Models para Testar

```
# backend/app/models/
- user.py (User model)
- account.py (Account model)
- transaction.py (Transaction model)
- category.py (Category model)
- budget.py (Budget model)
```

Comando do Agente

```
@workspace você é meu Model Testing Agent para FinanceFlow.

Para CADA model em backend/app/models/:

1. ANÁLISE DO MODEL:
   - Liste todos os campos e tipos
   - Identifique relacionamentos
   - Encontre validações e constraints
   - Mapeie métodos customizados

2. CRIE FIXTURES (tests/fixtures/):
   - fixtures/users.json - 10 usuários realistas
   - fixtures/accounts.json - 20 contas variadas
   - fixtures/transactions.json - 100 transações
   - fixtures/categories.json - categorias padrão
   - fixtures/budgets.json - orçamentos mensais

   Dados devem ser realistas:
   - Nomes brasileiros
   - CPFs válidos
   - Valores em R$
   - Datas coerentes

3. TESTES UNITÁRIOS (tests/models/):
   Para cada model, teste:
   - Criação com dados válidos
   - Validações de campos obrigatórios
   - Constraints únicos
   - Relacionamentos (1:N, N:N)
   - Métodos customizados
   - Soft delete se existir
   - Timestamps automáticos

4. TESTES DE INTEGRAÇÃO:
   - User com múltiplas Accounts
   - Account com Transactions
   - Transaction com Category
   - Budget validation rules
   - Cascading deletes

Use pytest com SQLAlchemy test database.
Coverage mínimo: 95%
```

Entregáveis

- [ ] Fixtures JSON com dados realistas brasileiros
- [ ] Testes para todos os 5 models
- [ ] Testes de relacionamentos
- [ ] Coverage report > 95%

EXERCÍCIO 4: AGENTE DOCUMENTADOR DE ROLES

Nível: Iniciante | Tempo: 20 minutos

Contexto

A pasta `/agents/` tem documentação de roles mas está incompleta. Melhorar e padronizar toda documentação.

Arquivos Existentes

```
agents/
- backend_developer.md
- database_administrator.md
- frontend_developer.md
- infra_quality.md
- product_owner.md
- tech_lead.md
- ux_designer.md
```

Comando do Agente

```
@workspace você é meu Team Documentation Agent.

Para CADA arquivo em /agents/:

1. PADRONIZE A ESTRUTURA:
   # Role: [Nome do Cargo]

   ## □ Responsabilidades Principais
   - Lista com 5-8 responsabilidades

   ## □ Stack Técnica
   - Ferramentas específicas do role
   - Tecnologias que precisa dominar

   ## □ KPIs e Métricas
   - Como medir sucesso neste role
   - Métricas quantitativas

   ## □ Interação com Outros Roles
   - Com quem colabora
   - Dependências
   - Entregas esperadas

   ## □ Checklist Diário
   - [ ] Tarefas recorrentes
   - [ ] Verificações importantes

   ## □ Objetivos do Sprint
   - Curto prazo (2 semanas)
   - Médio prazo (3 meses)

   ## □ Recursos e Referências
   - Links úteis
   - Documentação
   - Ferramentas

2. ADICIONE CONTEXTO FINANCEFLOW:
   - Específico para sistema financeiro
   - Compliance e segurança
   - Dados sensíveis

3. CRIE agents/README.md com:
   - Matriz RACI de responsabilidades
   - Fluxo de comunicação
   - Cerimônias e reuniões
   - Diagrama de interação entre roles

4. GERE agents/onboarding/:
   - Checklist por role
   - Primeiras tarefas
   - Recursos de aprendizado
```

Entregáveis

- [ ] 7 arquivos de roles padronizados
- [ ] README.md com matriz RACI
- [ ] Pasta onboarding/ com checklists
- [ ] Diagramas de interação

□ EXERCÍCIO 5: AGENTE DE MIGRAÇÃO E ATUALIZAÇÃO

Nível: Avançado | Tempo: 35 minutos

Contexto

O projeto precisa migrar de Pydantic v1 para v2 e atualizar React 18. Use agentes coordenados.

Estado Atual

```
Backend:
- Pydantic 1.10
- FastAPI 0.95
- SQLAlchemy 1.4

Frontend:
- React 18.2
- TypeScript 4.9
- Vite 4.0
```

Comando de Orquestração

```
@workspace você é meu Migration Orchestrator.

Coordene 3 agentes para atualizar FinanceFlow:

═══════════════════════════════════════
AGENTE 1: BACKEND MIGRATION SPECIALIST
═══════════════════════════════════════
Tarefas:
1. Analise todos schemas em backend/app/schemas/
2. Liste breaking changes Pydantic v1 → v2:
   - Config class → model_config
   - .dict() → .model_dump()
   - .json() → .model_dump_json()
   - Schema → BaseModel
   - Field validators changes

3. Para CADA arquivo:
   - Mostre código atual
   - Aplique migração
   - Mostre código migrado
   - Teste que funciona

4. Atualize requirements.txt:
   - pydantic>=2.0
   - pydantic-settings>=2.0
   - fastapi>=0.100

5. Crie migration_report.md com:
   - Arquivos modificados
   - Changes aplicadas
   - Potenciais issues

Output: "✅ Backend migrado para Pydantic v2"

═══════════════════════════════════════
AGENTE 2: FRONTEND UPDATE SPECIALIST
═══════════════════════════════════════
Após Agente 1:

1. Analise frontend/package.json
2. Identifique dependências desatualizadas
3. Para React e principais libs:
   - Check breaking changes
   - Update para latest stable
   - Fix deprecated warnings

4. Atualize:
```

```
4. Atualize:
   - TypeScript → 5.x
   - Vite → 5.x
   - React Router → 6.x
   - Redux Toolkit → latest

5. Refatore código afetado:
   - useEffect cleanup
   - Strict mode issues
   - TypeScript errors

Output: "✅ Frontend atualizado"

═══════════════════════════════════
AGENTE 3: TESTING VALIDATOR
═══════════════════════════════════
Após Agente 2:

1. Execute todos os testes:
   - Backend: pytest
   - Frontend: vitest
   - E2E: playwright

2. Para testes falhando:
   - Identifique causa
   - Corrija código ou teste
   - Re-execute

3. Valide integração:
   - Frontend chama backend
   - Auth funciona
   - CRUD operations OK

4. Gere test_report.md:
   - Coverage antes/depois
   - Testes corrigidos
   - Performance impact

Output: "✅ Todos testes passando"

RELATÓRIO FINAL com todas mudanças.
```

Entregáveis

- [ ] Backend migrado para Pydantic v2
- [ ] Frontend com dependências atualizadas
- [ ] Todos os testes passando
- [ ] Relatórios de migração detalhados
- [ ] Zero breaking changes para usuários

Validação Final

```
 # Backend
cd backend && pytest --cov=app

# Frontend
cd frontend && npm test

# E2E
npm run test:e2e

# Verificar app funcionando
docker-compose up
```

□ CRITÉRIOS DE AVALIAÇÃO GERAL

Para Cada Exercício

1. **Funcionalidade (40%)**
    - Código gerado funciona
    - Atende requisitos
    - Sem bugs críticos
2. **Qualidade (30%)**
    - Código limpo e organizado
    - Segue padrões do projeto
    - Bem documentado
3. **Completude (20%)**
    - Todas tarefas completadas
    - Arquivos no lugar correto
    - Testes incluídos
4. **Uso dos Agentes (10%)**
    - Comandos bem estruturados
    - Iterações para melhorar
    - Aprendizados documentados

Entrega

Para cada exercício, criar branch:

```
git checkout -b exercicio-1-api-docs
# Fazer mudanças
git add .
git commit -m "docs: add complete API documentation with agent"
git push origin exercicio-1-api-docs
```

DICA FINAL: Use o projeto real para aprender! Os agentes funcionam melhor quando têm contexto completo. Não hesite em iterar os comandos até obter o resultado desejado. □