



Aula 2 — Engenharia de Software e Storytelling com Dados

Contexto: Vocês devem prototipar um Middleware de Transparência que consome uma API pública, limpa e valida os dados ruidosos e expõe rotas HTTP seguras e previsíveis. Este material é técnico, didático e objetivo — voltado para estudantes de Engenharia de Software e desenvolvedores iniciantes em Python interessados em APIs e qualidade de dados.

Objetivo da aula: projetar uma API em Flask que implemente healthcheck, resumo de dados e filtros dinâmicos; incluir validação robusta (pydantic ou validação manual), tratamento de erros, caching simples e testes automatizados para garantir integridade e sanidade.

Visão Arquitetural — Onde o Middleware se encaixa



Client

Consumidores (frontend, analistas, scripts) solicitam rotas /status, /data/summary e /data/filter/.



Middleware de Transparência

Camada principal: coleta dados externos, valida, transforma, aplica cache e expõe JSON limpo.



APIs Externas

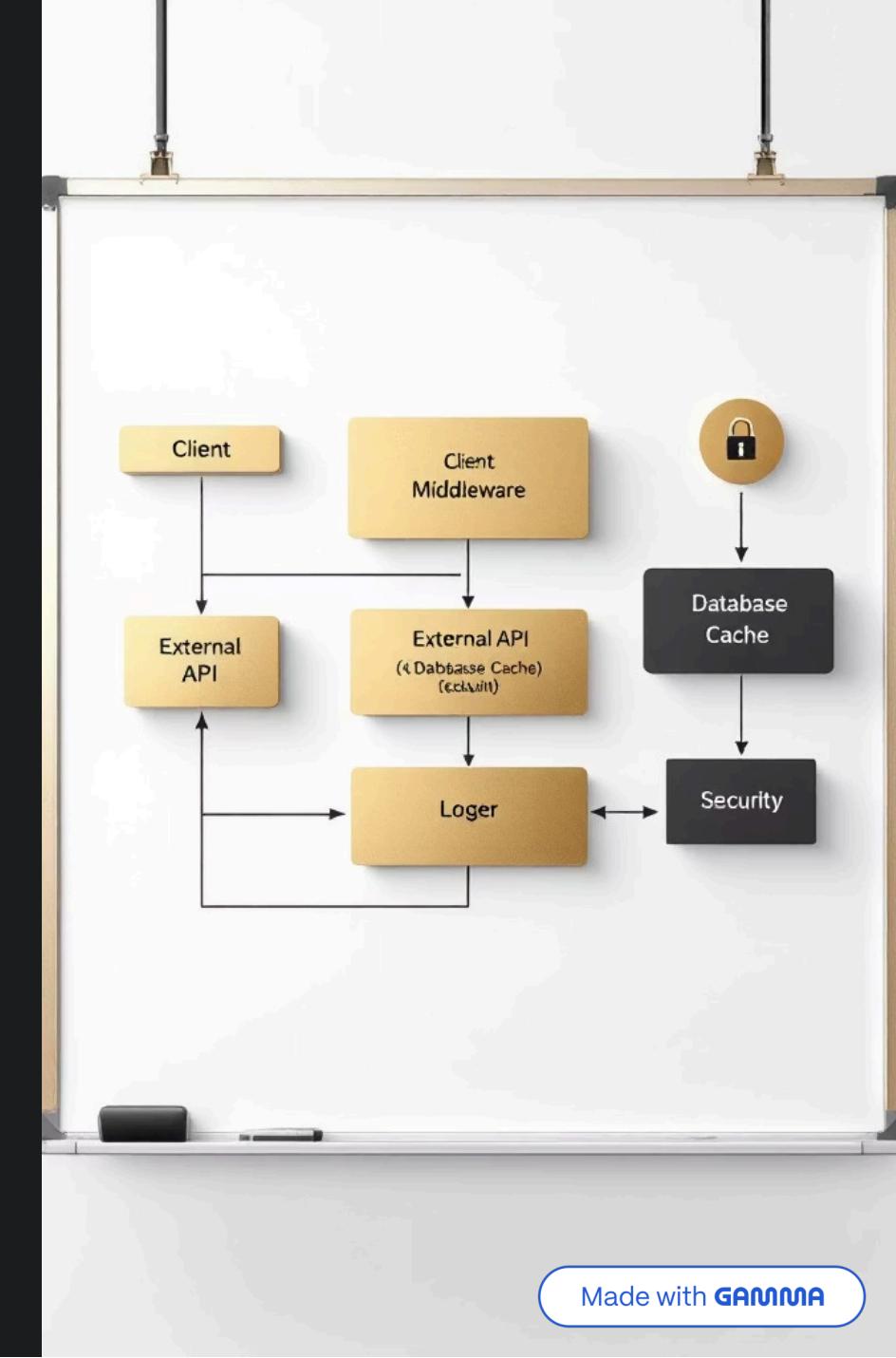
Fontes de dados abertos (ex: Câmara dos Deputados, OpenWeatherMap).
Potencial de dados ruidosos.



Observabilidade

Logs de integridade, métricas e tracing para auditar transformações e erros detectados.

Notas: mantenha separação de responsabilidades (rotas <-> validação <-> fetch <-> cache <-> transformação). Design orientado a contratos (schema-first) facilita testes e manutenção.





Requisitos de Rotas — Contratos da API

/status

Healthcheck simples: retorna uptime, versão da API, timestamp e status das dependências (cache & última fetch).

/data/summary

Consume fonte externa, aplica validações, calcula métricas agregadas (média, mediana, contagens por categoria) e retorna resumo com log de integridade.

/data/filter/<params>

Rota dinâmica para buscar por filtros (ex: data range, categoria, id). Deve suportar paginação e parâmetros de sanitização.

Importante: todos os endpoints retornam JSON padronizado com um campo meta.integrity_report contendo registros de campos inválidos, linhas descartadas e ações tomadas.

Validação e Integridade de Dados

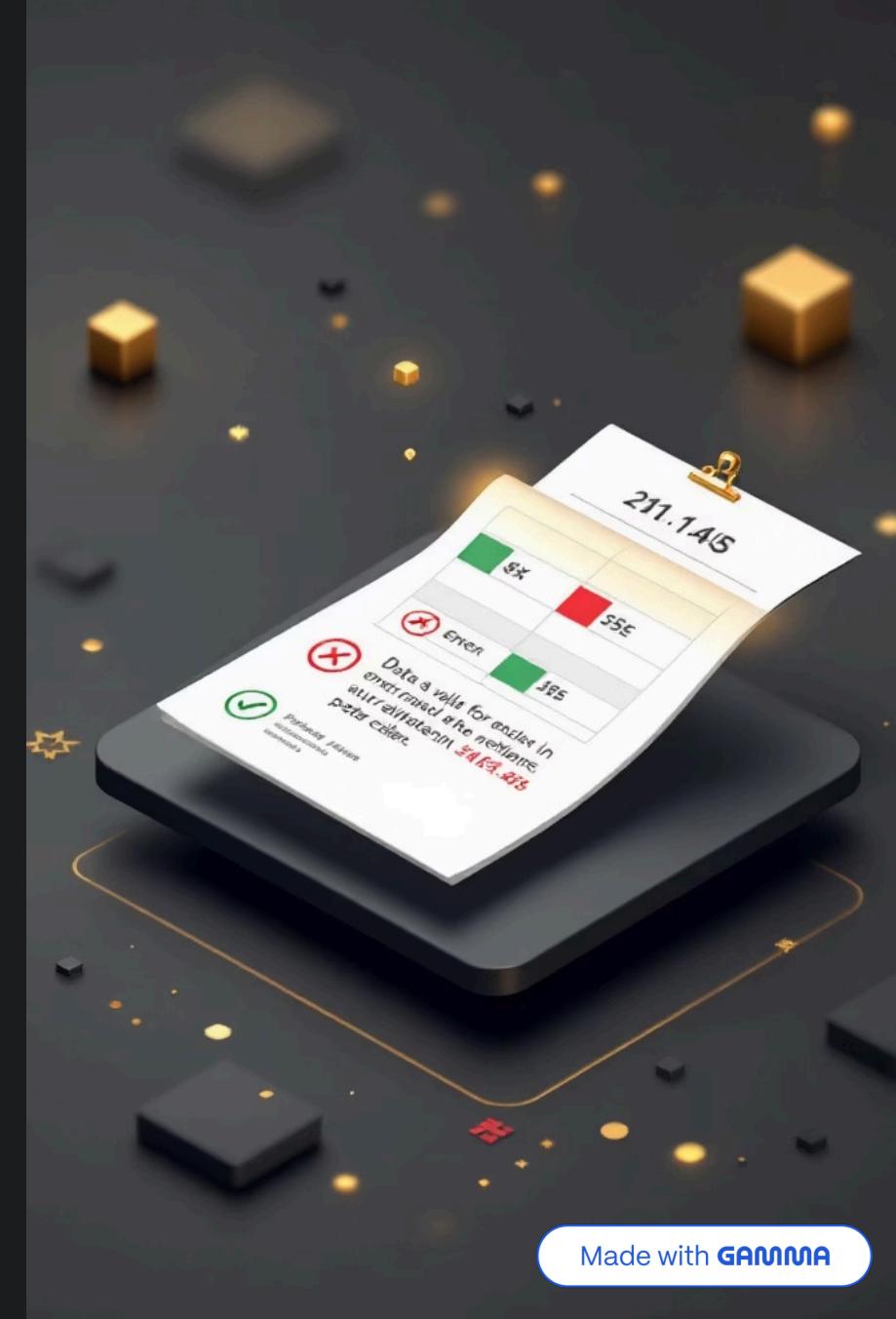
Princípios

- Contract-first: definir schemas antes de implementar transformação.
- Fail-safe: quando encontrar dados inválidos, não quebrar a API — registrar e retornar relatório.
- Sanitização: normalizar formatos de data, números e strings (ex: remover espaços, converter tipos).

Estratégias técnicas

- Uso recomendado: pydantic para validação e parsing automático (conversão de tipos, validação de formatos ISO 8601).
- Alternativa: decorators + validação manual para ambientes onde pydantic não é permitido.
- Validações específicas: preço ≥ 0 ; data em ISO 8601; ids obrigatórios; campos nulos marcados e tratados.

Exemplo de saída de integridade (meta.integrity_report): {total_rows, valid_rows, invalid_rows, errors_by_field: [...]}. Esse relatório é crítico para auditoria e storytelling com dados.



Testes Automatizados — O que validar

1. Status Code

Verificar que /data/summary e /data/filter/ retornam 200 em condições normais e retornam códigos apropriados (502/503) quando a fonte externa falha.

2. Schema Match

Testes que garantem presença de campos obrigatórios (id, valor, data_processamento, meta.integrity_report) e tipos esperados.

3. Sanidade Numérica e Temporal

Asserções: preço ≥ 0 ; datas em ISO 8601; valores fora de faixa devem ser descartados ou marcados no integridade_report.

Ferramentas sugeridas: pytest + responses ou requests-mock para simular respostas externas; coverage para medir alcance dos testes. Inclua testes para casos de borda (campos nulos, tipos trocados, paginador vazio).



Exemplo prático: Modelo pydantic e uso em Flask



Modelo Pydantic

Exemplo: class Record(BaseModel): id: int; price: condecimal(ge=0); date: datetime; category: Optional[str]

O pydantic faz parsing, valida e converte automaticamente strings de data para datetime.

Observação: capture ValidationError.exception para registrar quais campos falharam e exemplos de valores problemáticos — útil para feedback ao mantenedor da fonte de dados.



Integração na rota

Fluxo: fetch_external() → mapear e instanciar Record para cada item → coletar erros (ValidationError) → construir meta.integrity_report → retornar JSON com dados válidos e relatório.

Performance & Resiliência — Caching e Tratamento de Falhas

Caching Simples (em memória)

Implementação leve: LRU cache (`functools.lru_cache`) ou dicionário com TTL. Objetivo: reduzir chamadas a APIs externas durante picos e proteger o rate-limit da fonte.

Estratégia: cachear respostas agregadas por chave de consulta + expiry configurável (ex: 60s-300s dependendo da volatilidade).

Fallbacks e Circuit Breaker

Se a fonte externa falhar: retornar última resposta válida do cache com aviso no `meta.integrity_report` e status HTTP 200 (se aceitável) ou 503 quando nada estiver disponível.

Adotar backoff exponencial em novas tentativas e monitorar métricas de erro.

Checklist de Resiliência

- Timeouts explícitos em requests (ex: 5s)
- Retries com backoff limitado
- Cache com TTL e chave determinística
- Logs e métricas de integridade e latência

Storytelling com Dados — Como apresentar o relatório de integridade



Elementos do relatório

meta.integrity_report deve conter: total_rows, valid_rows, invalid_rows, errors_by_field (mapa campo -> contagem e exemplos), last_fetched_at, source_url e actions_taken.

Dica prática: inclua amostras (exemplos) de registros inválidos no relatório com hashes/anônimos para preservar privacidade, permitindo investigação posterior.



Narrativa

Construa uma narrativa clara: "dos N registros recuperados, X foram válidos, Y foram descartados por preço negativo, Z apresentaram data inválida". Use isso para priorizar correções na fonte e demonstrar transparência.

Exemplo de Estrutura de Repositório e Scripts de Teste



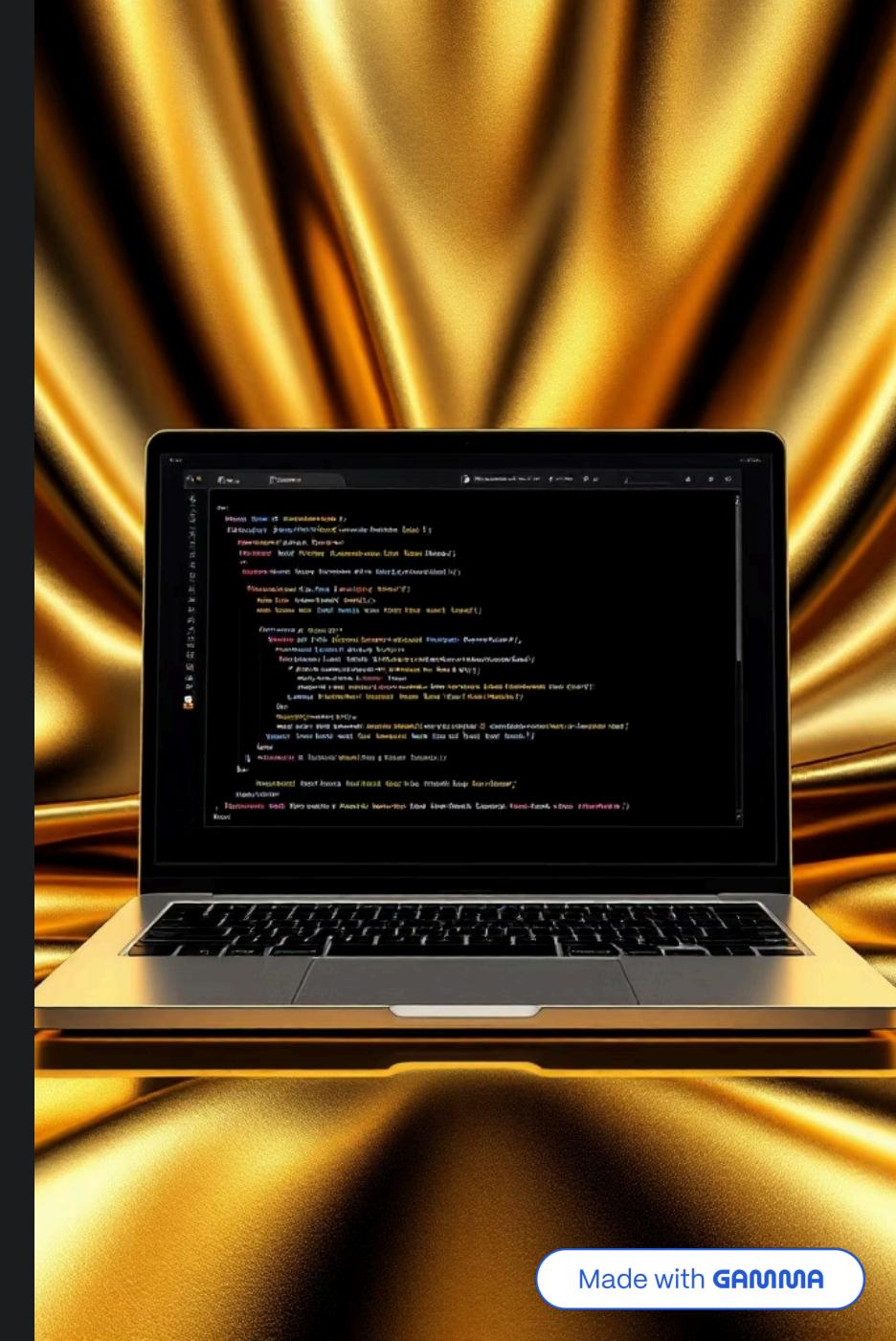
Estrutura sugerida

```
/app
├── main.py (Flask app)
├── fetcherpy (responsável por
  requests externas)
  ├── models.py (pydantic models)
  ├── validators.py (regras custom)
  ├── cache.py (TTL cache)
└── tests/ (pytest)
```



Conteúdo dos testes

- test_status_ok: checa /status e uptime.
- test_summary_schema: valida presença de id, valor, data_processamento e meta.integrity_report.
- test_price_non_negative: injeta registros com preço negativo e espera que sejam reportados no integrity_report.
- test_external_down: simula 503 da fonte externa e valida fallback do cache.



Ferramentas extras: CI (GitHub Actions) para rodar pytest em PRs; adicionar linters (flake8, black) e type checking (mypy) para garantir qualidade do código.

Resumo & Próximos Passos

1. Definir contratos (schemas)

Comece definindo pydantic models e exemplos válidos/inválidos para cada fonte de dados.

2. Implementar fetch + validação

Desenvolva funções que consumam a API externa, apliquem validações e gerem o `meta.integrity_report`.

3. Cache e Resiliência

Adicione caching em memória com TTL, timeouts e retry/backoff; trate erros corretamente.

4. Testes e Observabilidade

Escreva testes automatizados cobrindo status codes, schema e sanidade numérica; exponha logs e métricas.

5. Entrega e Documentação

Documente endpoints, contratos e políticas de integridade; prepare um protótipo para demonstração (Postman collection ou Swagger).

- Dica final: comece pequeno (um endpoint de fonte confiável) e itere — "start small, measure often, iterate quickly". A robustez do middleware vem da disciplina nos contratos e dos testes automatizados.