INSTITUTO FEDERAL
Paraná
Campus Paranaguá

TECNÓLOGO EM ANÁLISE E DESENVOLVIMENTO

Desenvolvimento Web III

Prof. Luiz Efigênio

Disciplina: Desenvolvimento Web III
Professor: Luiz Efigênio
Data: 03/10/2025

Introdução

Nesta sequência de aulas vamos consolidar a construção de APIs com Ruby on Rails e estudar em profundidade o estilo arquitetural REST, práticas para construir APIs RESTful robustas e como consumir APIs externas com segurança e performance. O objetivo é que vocês saiam capazes de projetar, implementar, testar e integrar APIs em projetos reais, entendendo escolhas arquiteturais, serialização, versionamento, autenticação e boas práticas operacionais.

Parte I — Ruby on Rails como API: fundamentos e serialização

Objetivos específicos

- Entender a diferença entre uma aplicação Rails tradicional e uma API-only.
- Conhecer as convenções do Rails para rotas, controllers e responses JSON.
- Aprender técnicas de serialização (ActiveModel::Serializers, Jbuilder, fast_jsonapi, JSON:API).
- Aplicar versionamento, error handling e políticas de caching específicas para APIs.

1. Rails API — conceitos e configuração

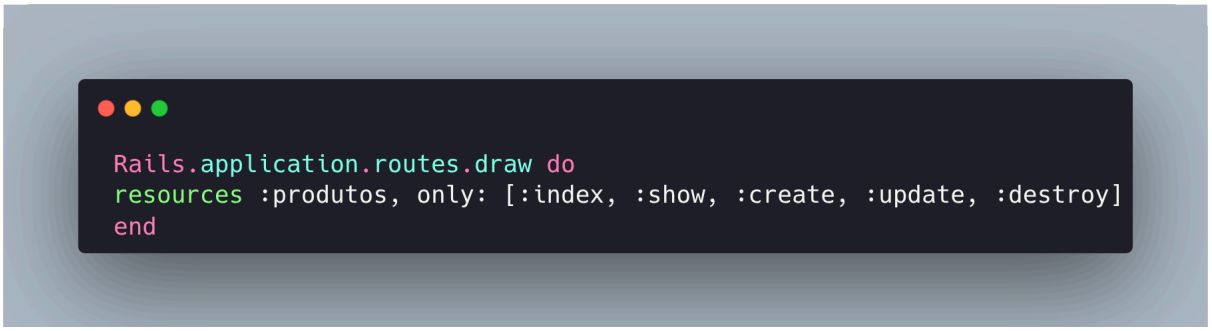
- Rails pode rodar em modo *API-only* reduzindo middleware e carregamento desnecessário, ideal para microservices ou backend para SPAs/mobile.

Criação de projeto API-only:

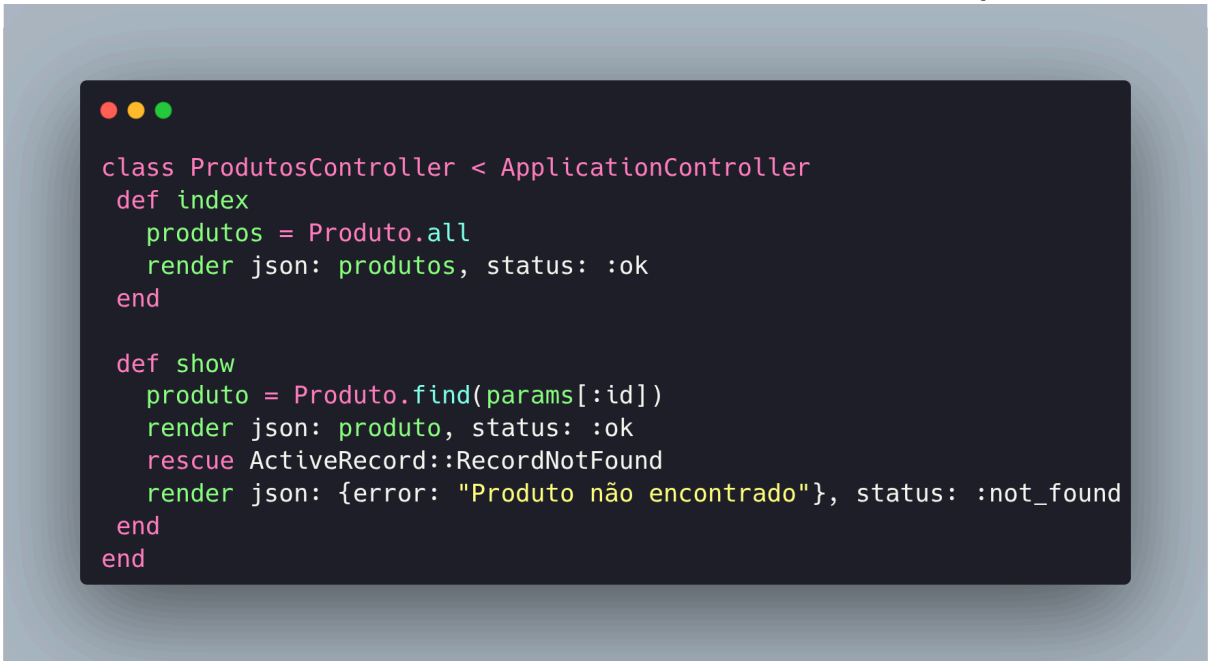
- Diferenças principais do template `--api`:
    - Middleware reduzido (menos sessions, view rendering).
    - Gemfile enxuto; resposta por padrão em JSON.
    - Foco em controllers que herdam `ActionController::API` em vez de `ActionController::Base`.

2. Convenções de rota e controller

- Rotas RESTful com `resources`:



- Controllers: ações respetivas devem devolver status HTTP apropriados e JSON padronizado:



```ruby
class ProdutosController < ApplicationController
 def index
   produtos = Produto.all
   render json: produtos, status: :ok
 end

 def show
   produto = Produto.find(params[:id])
   render json: produto, status: :ok
   rescue ActiveRecord::RecordNotFound
   render json: {error: "Produto não encontrado"}, status: :not_found
 end
end
```

3. Serialização — por que, quando e como

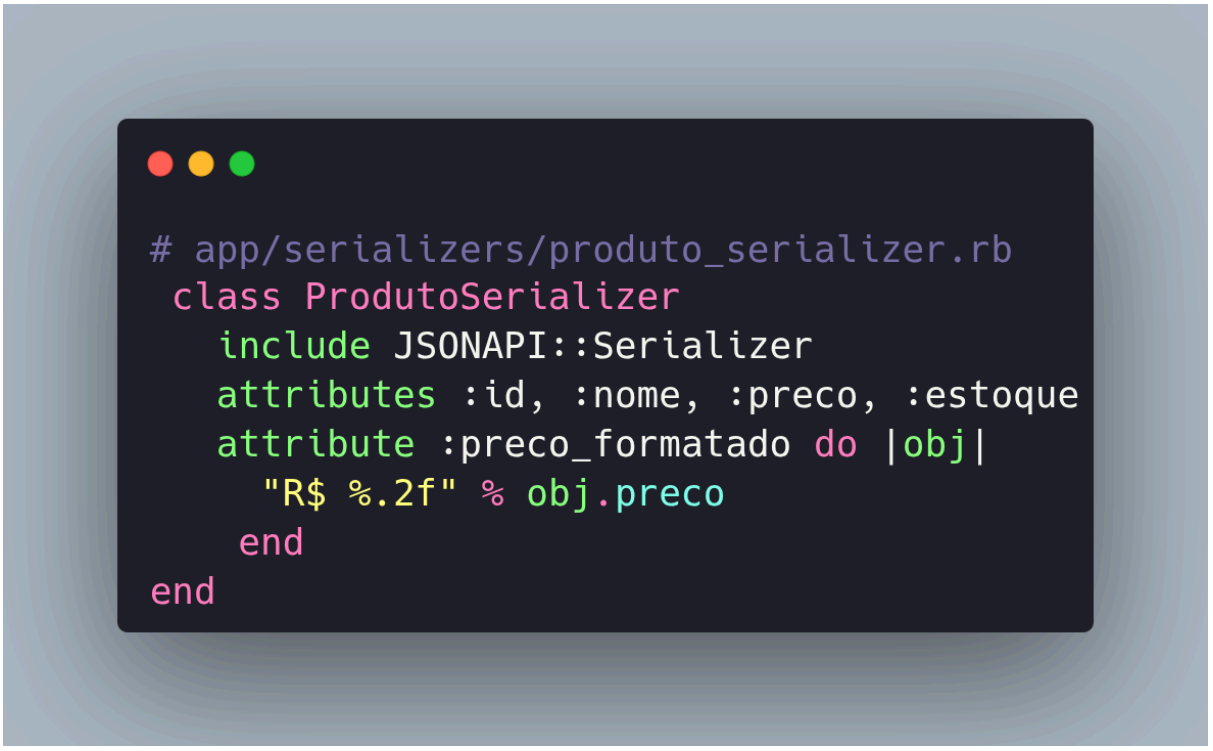
- *Por que serializar?* Para controlar shape do JSON, evitar expor campos sensíveis, reduzir payloads, e otimizar relacionamento (N+1).

- *Opções comuns:*

    - `to_json` / `as_json` (simples, rápido, mas pouco expressivo).
    - **Jbuilder** — templates JSON estruturados (boas para respostas customizadas).
    - **ActiveModel::Serializers (AMS)** — classes serializer para cada model.
    - **fast_jsonapi / jsonapi-serializer** — implementações otimizadas para especificação JSON:API (RC/mais rápido).
    - **JSON:API spec** (opcional) — padroniza `data`, `attributes`, `relationships`, `meta`, links e erros.

- *Recomendação prática:* para projetos didáticos e produção pequena: `fast_jsonapi` ou `jsonapi-serializer` por performance; Jbuilder quando a resposta é muito customizada.

4. Exemplo de serializador (ActiveModel::Serializer style)



```ruby
# app/serializers/produto_serializer.rb
 class ProdutoSerializer
   include JSONAPI::Serializer
   attributes :id, :nome, :preco, :estoque
   attribute :preco_formatado do |obj|
     "R$ %.2f" % obj.preco
    end
end
```

No controller:

```ruby
render json: ProdutoSerializer.new(produtos).serializable_hash, status: :ok
```

5. Evitando N+1 e controlando relacionamento

- Use `includes` para eager loading:

```ruby
produtos = Produto.includes(:categoria, :imagens).all
```

- No serializer, controle se relacionamentos devem ser embutidos ou links (HATEOAS).

6. Versionamento de API

- Estratégias:
    - Versionamento em rota: `/api/v1/produtos` (mais explícito).
    - Versionamento via header: `Accept: application/vnd.minhaapi.v1+json`.
    - Versionamento via subdomínio: `v1.api.exemplo.com`.
- Recomendo rota por ser mais simples para estudantes e fácil de controlar com namespaces:

```
namespace :api do
 namespace :v1 do
     resources :produtos
 end
end
```

7. Tratamento de erros e padrão de resposta

- Padronize estrutura de erro:

```
{
  "errors": [
    {"status":"404","title":"Not Found","detail":"Produto não encontrado"}
  ]
}
```

- Use `rescue_from` em `ApplicationController` para centralizar:

```ruby
class ApplicationController < ActionController::API
 rescue_from ActiveRecord::RecordNotFound do |e|
 render json: {errors: [{status: '404', title: 'Not Found', detail: e.message}]},
  status: :not_found
 end
end
```

8. Status HTTP e semântica

- `200 OK`, `201 Created` (com `Location` header para novo recurso), `204 No Content` (delete success), `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `422 Unprocessable Entity` (validações), `500 Internal Server Error`.

- Em `create` use `status: :created` e inclua URI do recurso (convenção RESTful).

9. Autenticação e autorização

- Autenticação em APIs: tokens JWT, OAuth2, API keys.
- Authorization: Pundit/CanCanCan (roles) para controlar recursos.
- Nunca enviar segredo em resposta; use `HttpOnly` cookie ou header `Authorization: Bearer <token>` dependendo do cliente.

10. Boas práticas e anti-patterns

- Não retorne dados sensíveis (senhas, tokens).
- Não exponha estruturas internas (ids de terceiros) sem controle.
- Evite verbos verbosos nas rotas (use recursos, não ações: `/produtos/1/ativar` → prefira `PATCH /produtos/1` com payload).
- Documente a API (OpenAPI / Swagger).

Parte II — APIs RESTful e consumo de APIs externas

Objetivos específicos

- Entender REST como estilo arquitetural e quais princípios realmente importam.
- Aprender desenho de recursos, versionamento, HATEOAS, idempotência e segurança.
- Conhecer bibliotecas Ruby para consumo de APIs externas e práticas robustas (timeouts, retries, circuit breaker).
- Praticar integração com serviços externos (pagamentos, mapas, APIs públicas).

1. REST e RESTful — fundamentos

- REST (Representational State Transfer) é um *estilo arquitetural* definido por Roy Fielding (dissertação, 2000). Não é um padrão rígido, mas um conjunto de restrições:
    - **Client–Server**: separação de responsabilidades.
    - **Statelessness**: cada requisição contém toda informação necessária.
    - **Cacheable**: respostas devem definir se são cacheáveis.
    - **Uniform interface**: identificação de resources (URIs), manipulação por representações, self-descriptive messages, HATEOAS.
    - **Layered system**: intermediários (proxy, load balancer).
- *RESTful* refere-se a serviços que seguem essas restrições em maior ou menor grau.

2. Design de recursos e URIs

- Nomeie recursos no plural: `/api/v1/produtos`.
- Use hierarquia para relacionamentos: `/clientes/:cliente_id/pedidos`.
- Use query params para filtros/pagination/sorting: `/produtos?categoria=eletronicos&page=2&per=20&sort=-preco`.
- Evite verbs na URI; use métodos HTTP.

3. Métodos HTTP e semântica

- `GET` — recuperar recursos; idempotente; cacheável.
- `POST` — criar recurso; não idempotente.
- `PUT` — substituir recurso; idempotente.
- `PATCH` — modificar parcialmente; semântica de alteração parcial.
- `DELETE` — remover; idempotente (repetir `DELETE` deve resultar no mesmo estado).

4. Idempotência e segurança

- Idempotência facilita retries seguros. Para operações não idempotentes use idempotency keys (ex.: `Idempotency-Key` header em APIs de pagamento).
- Segurança: TLS obrigatório, validação de entradas, rate limits e proteção contra SSRF/XSS/Injection.

5. HATEOAS

- HATEOAS (Hypermedia As The Engine Of Application State) propõe que responses incluam links para ações relacionadas:

```
{
  "data": {...},
  "links": {"self": "/api/v1/produtos/1", "comprar": "/api/v1/produtos/1/compra"}
}
```

- Na prática muitos serviços não implementam HATEOAS por completo, mas fornecer links úteis melhora a usabilidade.

6. Paginação, filtragem e ordenação

- Padrões comuns de paginação:
    - Page-based: `page` + `per_page`.
    - Cursor-based: `cursor=abcdef` (melhor para grandes datasets e APIs públicas).
- Inclua meta com total, page, per_page.
- Filtragem por query params; evitar envio de filtros complexos via body de GET.

7. Documentação e contratos

- OpenAPI / Swagger para documentar endpoints, schemas, códigos de resposta e exemplos.
- Contratos claros evitam quebra na integração (backwards compatibility).
- Versionamento quando mudanças quebram compatibilidade.

8. Consumo de APIs externas em Ruby/Rails

- Bibliotecas comuns:
    - `Net::HTTP` — stdlib (baixo nível).
    - `HTTParty` — simples e expressivo.
    - `Faraday` — adaptável com middleware (recommended).
    - HTTP gem (http.rb) — performática e moderna.
- Recomendação: usar `Faraday` por flexibilidade e suporte a middlewares (timeout, retry, logging).

9. Padrões de integração robustos

- Timeouts: sempre configurar `open_timeout` e `read_timeout`.
- Retries: backoff exponencial, com jitter; limitar número de tentativas.
- Circuit breaker: proteger serviços que falham repetidamente (gem `circuitbox`).
- Bulkheads: isolar chamadas externas para não derrubar todo sistema.
- Fallback: respostas padrão ou cache quando serviço externo falha.

10. Exemplo prático com Faraday

```ruby
connection = Faraday.new(url: 'https://api.externa.com') do |f|
 f.request :json
 f.response :json, content_type: /\bjson$/
 f.options.timeout = 5
 f.options.open_timeout = 2
 f.adapter Faraday.default_adapter
 end
resp = connection.get('/v1/recursos', {q: 'termo'})
 if resp.success?
 data = resp.body
 else
 Rails.logger.error("Erro API externa: #{resp.status}")
end
```

11. Autenticação em consumo de APIs

- API Keys (header `Authorization: Token abc` ou `x-api-key`).
- OAuth2 (client credentials / authorization code). Gems: `oauth2`.
- JWT exchange: receber token de serviço e usar Bearer.
- Armazenar segredos em credenciais (`rails credentials:edit`) ou Vault; nunca em repo.

12. Tratamento de erros e segurança

- Logue erros com contexto (request id, endpoint) mas nunca registre segredos.
- Verifique status code: 2xx success, 4xx client error (não reverter/retry), 5xx server error (retry com cautela).
- Proteja contra SSRF validando URLs de callbacks e host allowlists.

13. Observabilidade

- Tempo de resposta, taxa de erros, percentil P95/P99 em métricas (Prometheus + Grafana).
- Distributed tracing (OpenTelemetry) para observar chamadas entre serviços.

14. Exemplo de fluxo real (pagamentos)

- Recebe pedido -> cria `payment_intent` via API do provedor -> provedor retorna `id` -> cliente conclui -> callback (webhook) confirma -> sistema atualiza pedido.
- Implemente verificação de firma no webhook e confirme idempotência (evitar processar webhook duplicado).

15. Documentação e contrato

- Publicar OpenAPI + exemplos de request/response + autenticação.
- Fornecer changelog de breaking changes por versão.

Atividade prática

Parte A — Implementação (Rails API)

1. Individual, crie um microservice Rails API-only `minha_loja_api` com namespace `/api/v1`.
2. Modele duas entidades: `Produto` e `Pedido` com relacionamento `Pedido has_many :itens`.
3. Implemente endpoints RESTful para `produtos` e `pedidos`.
4. Use serializadores (`jsonapi-serializer` ou `fast_jsonapi`) para formatar saída.

Parte B — Integração com API externa

1. Consuma uma API pública (ex.: API de CEP, ou API de câmbio) usando `Faraday`.
2. Implemente timeouts e retries com backoff.
3. Simule falha externa e implemente fallback (resposta amigável).
4. Documente em OpenAPI a rota que integra com serviço externo.

Entrega

- Código no repositório GitHub (link no classroom).
- README descrevendo endpoints, autenticação de testes (se aplicável) e como rodar localmente.
- Relatório (máx 2 páginas) explicando decisões de design (serialização, versionamento, caching, retries).

Reflexão

APIs são o tecido conectivo das aplicações modernas. Saber construir uma API bem desenhada e integrar-se com serviços externos de forma robusta, segura e observável é habilidade central de qualquer desenvolvedor web atual. Nesta disciplina, queremos que você combine boas práticas arquiteturais com pragmatismo: escolha padrões que favoreçam manutenção, performance e segurança, e documente suas decisões para facilitar evolução do sistema.

Referências

- Fielding, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures.* (dissertação, 2000).
- Richardson, Leonard; Ruby, Mike. *RESTful Web APIs.* O'Reilly.
- JSON:API — https://jsonapi.org
- Faraday README / HTTParty docs / Net::HTTP ruby docs
- OpenAPI Specification — https://swagger.io/specification/
- OWASP API Security Top 10 — https://owasp.org
- Rails Guides — *Creating an API* (https://guides.rubyonrails.org)
- JSON:API Spec — https://jsonapi.org
- jsonapi-serializer / fast_jsonapi gem docs
- "Rails: Up and Running" / "Agile Web Development with Rails" (capítulos sobre controllers & rendering)