Sistema de Web Scraping para Coleta e Disponibilização de Dados do Mercado Imobiliário

Allan Knecht

Curso de Ciência da Computação – Universidade Regional Integrada do Alto Uruguai e das Missões (URI)
Av. Sete de Setembro, 1621 – 99709-910 – Erechim – RS – Brasil

046210@aluno.uricer.edu.br

Abstract. *This paper presents a web scraping system for automated collection and availability of real estate market data. The system architecture separates backend and frontend responsibilities, using Ruby on Rails for data collection, normalization and API exposure, and Vue.js for user interface. The solution implements asynchronous scraping techniques, RESTful APIs, JWT authentication, and Docker containerization to ensure scalability, security and reproducibility across different environments.*

Resumo. *Este trabalho apresenta um sistema de web scraping para coleta automatizada e disponibilização de dados do mercado imobiliário. A arquitetura do sistema separa as responsabilidades entre backend e frontend, utilizando Ruby on Rails para coleta, normalização e exposição dos dados via API, e Vue.js para interface do usuário. A solução implementa técnicas de scraping assíncrono, APIs RESTful, autenticação JWT e containerização Docker para garantir escalabilidade, segurança e reprodutibilidade em diferentes ambientes.*

1. Introdução

2. Referencial Teórico

O desenvolvimento de aplicações web modernas evoluiu de soluções monolíticas para arquiteturas distribuídas, que separam a camada de apresentação (frontend) da lógica de negócios e persistência de dados (backend). Essa separação aumenta a escalabilidade, a manutenibilidade e a clareza da solução. Segundo Sommerville (2011), a modularização em sistemas de software é um dos pilares para reduzir a complexidade em projetos de longo prazo.

2.1. Web Scraping

O web scraping é uma técnica que possibilita a extração automatizada de dados disponíveis em páginas web. Mitchell (2018) explica que o scraping envolve três estágios principais: a obtenção do conteúdo da página (fetching), a análise da estrutura HTML (parsing) e a extração dos dados relevantes (extraction). Essa prática é amplamente utilizada em sistemas de monitoramento de preços, agregadores de conteúdo e pesquisas acadêmicas que dependem de coleta massiva de dados públicos.

Uma das principais dificuldades no scraping é a heterogeneidade do HTML entre diferentes sites. Cada portal pode empregar estruturas e classes distintas, exigindo que o desenvolvedor crie estratégias adaptadas de parsing. Segundo Black (2016), o uso

de bibliotecas especializadas como Nokogiri em Ruby permite percorrer o DOM de maneira eficiente, oferecendo expressões CSS e XPath para localizar informações.

O desafio adicional é a manutenção dos scrapers. Mudanças frequentes nos layouts dos sites podem inviabilizar regras de parsing previamente definidas, exigindo ajustes constantes. Nesse sentido, a literatura aponta a necessidade de adotar arquiteturas de scraping flexíveis, que desacoplem a coleta dos detalhes de parsing (Mitchell, 2018).

Outra questão importante no web scraping é a ética e a legalidade do processo. Mitchell (2018) lembra que a coleta de dados deve respeitar os termos de uso dos sites, evitando sobrecarga dos servidores e garantindo conformidade com a legislação vigente. Por isso, práticas como a utilização de atrasos entre requisições, limitação de taxa de acesso e identificação clara do agente de coleta são recomendadas. Essas estratégias não apenas preservam a infraestrutura das fontes, mas também reforçam a credibilidade do sistema desenvolvido.

2.2. Arquitetura e Backend com Ruby on Rails

O Ruby on Rails é um framework que segue a arquitetura Modelo-Vista-Controlador (MVC), consolidando boas práticas de organização de software (Fowler, 2002). Sua filosofia "convention over configuration" reduz a quantidade de código necessária, privilegiando convenções pré-estabelecidas para tarefas recorrentes (Hartl, 2016).

No Rails, o componente Active Record implementa o padrão de Object-Relational Mapping (ORM), que permite mapear tabelas do banco de dados em objetos da aplicação. Evans (2004) ressalta que a utilização de ORMs favorece o desenvolvimento orientado a domínio, evitando que desenvolvedores manipulem SQL bruto e possibilitando maior foco na lógica de negócio.

Outro recurso central do Rails são as *migrations*, que permitem versionar o esquema do banco de dados. De acordo com Hartl (2016), *migrations* possibilitam a evolução incremental da estrutura da base, garantindo consistência entre diferentes ambientes. Isso elimina a necessidade de manipulação manual de SQL, aproximando o gerenciamento de dados do fluxo ágil de desenvolvimento.

O backend desenvolvido para este trabalho adota o paradigma de APIs RESTful, o que permite que múltiplos clientes — como aplicações web ou móveis — consumam os dados extraídos. Fielding (2000) definiu o estilo arquitetural REST como baseado em recursos identificados por URIs e manipulados por meio de operações padronizadas (GET, POST, PUT, DELETE), promovendo interoperabilidade.

Além disso, a utilização de tarefas assíncronas para scraping periódicos é fundamental. Tate (2016) observa que a delegação de atividades custosas para filas de execução aumenta a responsividade de sistemas distribuídos, pois libera recursos da aplicação principal enquanto processos intensivos são realizados em segundo plano.

2.3. Frontend com Vue.js

No frontend, frameworks modernos têm buscado oferecer interfaces mais reativas e dinâmicas. O Vue.js é um dos mais adotados por sua proposta de ser progressivo e de fácil integração. Segundo You (2014), Vue combina conceitos de componentes reativos

e ligação bidirecional de dados (two-way binding), permitindo que a interface se atualize automaticamente quando os dados da aplicação são alterados.

A utilização de frameworks como Vue.js permite a construção de Single Page Applications (SPAs), em que a interação do usuário ocorre sem recarregamentos completos de página. Isso melhora a experiência e o desempenho da aplicação (Flanagan, 2020). Em projetos que envolvem consumo de APIs, como o presente, o Vue facilita a comunicação assíncrona com o backend via chamadas AJAX ou fetch/axios, tornando possível a implementação de recursos de busca, filtros dinâmicos e exibição de resultados em tempo real.

Além disso, a separação de responsabilidades entre frontend e backend proporciona maior flexibilidade de evolução tecnológica. Enquanto o backend pode se concentrar em scraping e disponibilização de dados via API, o frontend pode ser atualizado para oferecer novas interações e visualizações sem necessidade de alterar a lógica central do sistema (Sommerville, 2011).

2.4. Persistência de Dados

A persistência dos dados coletados foi realizada com PostgreSQL, um banco de dados relacional amplamente reconhecido pela sua confiabilidade e suporte a operações transacionais. Date (2004) defende que o modelo relacional continua sendo a base mais robusta para aplicações críticas, pois assegura integridade e consistência de dados.

O uso de ORMs como o Active Record contribui ainda para manter a consistência da aplicação, permitindo que os registros de imóveis sejam tratados como objetos Ruby, mas armazenados de forma estruturada em tabelas relacionais.

2.5. Segurança e Autenticação

Em sistemas que expõem dados via API, a segurança é um ponto essencial. A adoção de JSON Web Tokens (JWT) permite autenticação sem estado, em que o cliente armazena o token assinado e o apresenta a cada requisição subsequente. Jones et al. (2015) destacam que tokens assinados podem ser validados sem a necessidade de manter sessões em banco, o que aumenta a escalabilidade.

A escolha pelo JWT também está relacionada à simplicidade de integração com aplicações distribuídas. Diferente de sessões tradicionais armazenadas no servidor, o token é autossuficiente e pode conter informações como escopos de acesso ou permissões específicas (Jones et al., 2015). Isso permite que a API seja consumida de forma segura por múltiplos clientes, inclusive aplicações móveis e serviços de terceiros, sem a necessidade de replicar estado entre servidores. Essa característica torna a solução mais aderente a arquiteturas modernas de microsserviços.

No Rails, bibliotecas como Devise simplificam a integração de JWT, fornecendo suporte a estratégias de autenticação baseadas em denylist de tokens, fortalecendo a proteção contra acessos indevidos.

2.6. Infraestrutura e Deploy

Por fim, a utilização de Docker para containerização garante que o sistema possa ser executado em diferentes ambientes com reprodutibilidade. Merkel (2014) mostra que containers encapsulam dependências, bibliotecas e configurações, garantindo que a

aplicação funcione da mesma maneira em máquinas distintas, seja em desenvolvimento local ou em servidores de produção.

3. Metodologia

A metodologia adotada para o desenvolvimento do sistema é de natureza aplicada e experimental, com foco na construção de uma aplicação prática para coleta e disponibilização de dados do mercado imobiliário.

3.1. Estrutura do Sistema

O sistema foi planejado em duas camadas principais:

- Backend (Rails API): responsável pela coleta, normalização, persistência e disponibilização dos dados de imóveis.

- Frontend (Vue.js): responsável por consumir os endpoints da API e apresentar os dados em uma interface amigável, com recursos de busca e filtragem.

3.2. Coleta de Dados

A etapa inicial consistiu na identificação dos sites imobiliários relevantes. Para cada fonte, foi desenvolvido um scraper independente, capaz de percorrer páginas de listagem e extrair atributos essenciais, como preço, localização, número de dormitórios, área útil, entre outros.

Os scrapers foram configurados para realizar tanto a coleta superficial (listagens) quanto a coleta detalhada (páginas de cada imóvel), garantindo maior riqueza dos dados armazenados.

3.3. Normalização e Persistência

Após a extração, os dados foram normalizados e persistidos em um banco de dados relacional. Essa normalização foi necessária para tratar diferenças de formatação entre os diversos portais. O banco de dados escolhido foi o PostgreSQL, que suporta consultas complexas e oferece robustez transacional.

3.4. Exposição via API REST

Os dados coletados foram disponibilizados por meio de endpoints RESTful. Essa API permite que diferentes clientes possam acessar os dados de maneira padronizada, utilizando operações HTTP. Foram implementados endpoints de listagem, consulta por filtros e detalhamento de imóveis.

3.5. Autenticação e Segurança

Para assegurar que apenas usuários autorizados consumam a API, foi implementada autenticação baseada em JWT. Dessa forma, após autenticação inicial, o usuário recebe um token que deve ser enviado em cada requisição subsequente. Tokens inválidos ou revogados são bloqueados, reforçando a segurança.

3.6. Processamento Assíncrono

A execução dos scrapers foi delegada a tarefas assíncronas, permitindo que a coleta seja realizada de maneira periódica, sem comprometer o desempenho da API. Essa estratégia garante escalabilidade e evita bloqueios de requisições externas.

3.7. Frontend

O frontend foi desenvolvido em Vue.js, estruturado como uma Single Page Application (SPA). Ele se conecta aos endpoints da API e fornece ao usuário recursos de pesquisa e filtragem em tempo real. Foram implementados componentes reativos para exibição de listas de imóveis, visualização detalhada de registros e mecanismos de busca por critérios como localização, preço e número de dormitórios.

3.8. Containerização

Por fim, todo o sistema foi containerizado com Docker, permitindo que a mesma configuração de ambiente seja reproduzida em diferentes máquinas. Isso garante facilidade de deploy e reduz inconsistências entre ambientes de desenvolvimento e produção.

4. Desenvolvimento

5. Resultados

6. Discussão

7. Conclusões

Referências

Black, D. (2016). The Well-Grounded Rubyist. 2nd ed. Manning Publications.

Date, C. J. (2004). An Introduction to Database Systems. 8th ed. Addison-Wesley.

Evans, E. (2004). Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

Fielding, R. (2000). Architectural Styles and the Design of Network-based Software Architectures. Tese de Doutorado. University of California.

Flanagan, D. (2020). JavaScript: The Definitive Guide. 7th ed. O'Reilly Media.

Fowler, M. (2002). Patterns of Enterprise Application Architecture. Addison-Wesley.

Hartl, M. (2016). Ruby on Rails Tutorial: Learn Web Development with Rails. 4th ed. Addison-Wesley.

Jones, M.; Bradley, J.; Sakimura, N. (2015). JSON Web Token (JWT). RFC 7519. IETF. Disponível em: https://www.rfc-editor.org/rfc/rfc7519.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. Linux Journal, 2014(239), 2.

Mitchell, R. (2018). Web Scraping with Python: Collecting Data from the Modern Web. 2nd ed. O'Reilly Media.

Sommerville, I. (2011). Software Engineering. 9th ed. Addison-Wesley.

Tate, B. (2016). Seven Concurrency Models in Seven Weeks. Pragmatic Bookshelf.

You, E. (2014). Vue.js: The Progressive JavaScript Framework. Disponível em: https://vuejs.org/.