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

O mercado imobiliário brasileiro apresenta um comportamento notoriamente dinâmico, caracterizado por flutuações constantes nos preços, disponibilidade e nas características dos imóveis. Essa dinamicidade reflete fatores econômicos, sociais e urbanos que influenciam diretamente a demanda e a oferta de propriedades. Acompanhar tais oscilações tornou-se uma necessidade estratégica tanto para profissionais do setor, que buscam oportunidades de investimento, quanto para pesquisadores, que utilizam essas informações em análises de mercado e estudos urbanos.

Contudo, a fragmentação das fontes de dados — distribuídas em diversos portais imobiliários e plataformas de anúncios — torna a coleta manual uma tarefa impraticável. Além da heterogeneidade nos formatos e estruturas de apresentação, há o problema da atualização constante: imóveis são incluídos, removidos ou modificados a todo instante. Assim, a busca por métodos automatizados de coleta, padronização e disponibilização de informações torna-se não apenas conveniente, mas indispensável.

Nesse contexto, o presente trabalho propõe o desenvolvimento de um sistema de web scraping voltado à coleta automatizada de dados imobiliários, visando superar os desafios de heterogeneidade estrutural e volume de informações. A solução busca agregar dados de múltiplas fontes, transformando-os em um formato uniforme e acessível por meio de APIs RESTful.

O projeto adota uma arquitetura moderna e modular, baseada na separação entre backend e frontend, o que favorece escalabilidade, manutenibilidade e clareza de

responsabilidades. O backend, desenvolvido em Ruby on Rails, é responsável pela coleta, normalização e disponibilização dos dados, enquanto o frontend, implementado em Vue.js, fornece uma interface reativa e interativa para visualização e análise.

A escolha dessas tecnologias não é arbitrária. O Ruby on Rails é amplamente reconhecido pela sua produtividade, convenções e suporte nativo a APIs, além de integrar ferramentas maduras para manipulação de dados e tarefas assíncronas. Já o Vue.js se destaca por sua leveza, curva de aprendizado acessível e integração simples com REST APIs, permitindo construir Single Page Applications (SPAs) com desempenho elevado e responsividade aprimorada.

Adicionalmente, o sistema incorpora práticas contemporâneas de desenvolvimento, como autenticação baseada em JSON Web Tokens (JWT), processamento assíncrono e conteinerização com Docker, garantindo segurança, isolamento de ambientes e reprodutibilidade do processo de implantação. Essas características tornam o sistema facilmente adaptável a diferentes contextos, incluindo uso corporativo, acadêmico e de pesquisa.

Assim, o objetivo central deste trabalho é apresentar um sistema capaz de automatizar a coleta e organização de dados do mercado imobiliário brasileiro, fornecendo uma base confiável e extensível para análise e integração com outros serviços. O estudo busca demonstrar que, ao combinar tecnologias consolidadas com boas práticas de engenharia de software, é possível criar uma solução eficiente, escalável e sustentável para um problema real de grande relevância prática.

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 implementado um mecanismo de autenticação híbrido que combina o uso do Devise e do JSON Web Token (JWT). O Devise, amplamente utilizado no ecossistema Ruby on Rails, é responsável pelo gerenciamento completo de usuários, incluindo registro, autenticação inicial, redefinição de senhas e confirmação de contas. Já o JWT é utilizado para viabilizar sessões sem estado, dispensando o uso de cookies e tornando a comunicação entre cliente e servidor mais eficiente em ambientes distribuídos.

Após o processo de autenticação realizado pelo Devise, o sistema gera um token JWT assinado criptograficamente, que é enviado ao cliente. Esse token deve ser incluído no cabeçalho de cada requisição subsequente, servindo como comprovante de identidade. Dessa forma, o servidor pode validar rapidamente a autenticidade da requisição sem precisar armazenar sessões na memória, o que aumenta a escalabilidade e a segurança da API.

Além disso, tokens expirados, inválidos ou revogados são automaticamente rejeitados, impedindo acessos indevidos e reduzindo o risco de sequestro de sessão. Essa combinação entre Devise e JWT oferece um equilíbrio eficaz entre usabilidade e proteção, assegurando que apenas usuários devidamente autenticados possam consumir os recursos da aplicação.

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

O desenvolvimento do sistema foi conduzido de forma incremental e iterativa, seguindo princípios da engenharia de software ágil. Essa abordagem permitiu ajustar funcionalidades progressivamente, com base em testes e validações contínuas. O processo foi dividido em quatro fases principais: coleta, processamento, disponibilização e apresentação dos dados.

4.1 Backend e Coleta de Dados

A primeira etapa consistiu na implementação do backend em Ruby on Rails, estruturado como uma API independente. Essa separação elimina dependências entre as camadas de aplicação, tornando o sistema flexível e compatível com diferentes clientes, como interfaces web, aplicativos móveis ou scripts externos.

Cada scraper foi projetado como um serviço autônomo, responsável por extrair informações específicas de um portal imobiliário. Essa modularidade permite que novos scrapers sejam adicionados sem alterar o núcleo do sistema, bastando implementar classes que sigam a mesma interface de coleta. Para processar as páginas HTML, utilizou-se a biblioteca Nokogiri, que possibilita o parsing do DOM e a extração seletiva de informações com alta performance.

A principal dificuldade enfrentada nessa fase foi a heterogeneidade estrutural das páginas. Diferentes portais utilizam marcações HTML e convenções distintas, exigindo regras específicas de parsing. Para contornar esse problema, foi criada uma camada de normalização, responsável por padronizar unidades de medida, formatações numéricas e nomenclaturas. Por exemplo, valores monetários são convertidos para um

formato numérico uniforme, e áreas são expressas consistentemente em metros quadrados.

4.2 Processamento e Persistência

Após a coleta, os dados passam por um processo de validação e limpeza, removendo informações duplicadas ou inconsistentes. A persistência é feita no banco PostgreSQL, utilizando o Active Record como ORM (Object-Relational Mapping). Essa escolha permite que as entidades da aplicação sejam manipuladas como objetos Ruby, enquanto o framework se encarrega de gerar o SQL subjacente.

As migrations do Rails garantem versionamento e rastreabilidade do esquema do banco, o que é crucial para ambientes colaborativos. Dessa forma, qualquer modificação estrutural pode ser reproduzida com segurança em outros ambientes, mantendo consistência entre desenvolvimento, teste e produção.

4.3 API RESTful e Segurança

A camada de disponibilização de dados foi implementada segundo o estilo arquitetural REST, definido por Fielding (2000). A API oferece endpoints para listagem, filtragem e detalhamento de imóveis, utilizando os métodos HTTP padrão (GET, POST, PUT e DELETE).

A segurança é garantida por autenticação JWT, em que cada usuário autenticado recebe um token criptograficamente assinado. Esse token é enviado junto às requisições subsequentes, permitindo autenticação sem necessidade de armazenar sessões no servidor. Essa abordagem é mais escalável e compatível com aplicações distribuídas.

4.4 Frontend e Experiência do Usuário

O frontend, construído com Vue.js, foi planejado como uma Single Page Application (SPA). Essa arquitetura permite que o usuário navegue e filtre dados sem recarregar a página inteira, tornando a experiência fluida e contínua.

Foram desenvolvidos componentes reativos para busca, filtragem e visualização detalhada de imóveis. A comunicação com o backend ocorre via Axios, e o gerenciamento de estado é feito por meio de uma estrutura reativa simplificada. Essa estratégia reduz acoplamentos e melhora a legibilidade do código.

4.5 Conteinerização e Reprodutibilidade

Por fim, todo o sistema foi containerizado com Docker, assegurando que dependências, versões de bibliotecas e configurações sejam idênticas em qualquer ambiente. Arquivos

Dockerfile foram criados para backend e frontend, enquanto o docker-compose.yml orquestra a execução conjunta de serviços, incluindo banco de dados e filas de background. Essa abordagem garante implantação rápida, isolamento e reprodutibilidade total do sistema.

5. Resultados

Os resultados obtidos demonstram que o sistema desenvolvido atinge seus objetivos de automação, normalização e disponibilização de dados de forma eficaz. A aplicação foi capaz de extrair informações de múltiplos portais imobiliários, consolidando listagens e detalhes em um repositório unificado.

Os testes realizados mostraram que a execução assíncrona dos scrapers evita bloqueios e mantém a API disponível para usuários, mesmo durante coletas em segundo plano. Essa característica é essencial em sistemas distribuídos que demandam alta disponibilidade.

A interface web desenvolvida com Vue.js se mostrou intuitiva e responsiva. A utilização de filtros dinâmicos e atualização em tempo real aumentou significativamente a eficiência da consulta e a satisfação do usuário. A integração entre backend e frontend via API RESTful demonstrou excelente desempenho e interoperabilidade.

Outro resultado relevante é a redução do tempo de implantação obtida com a containerização. Em testes realizados em diferentes máquinas, o sistema pôde ser inicializado integralmente com um único comando, sem necessidade de configuração manual. Essa reprodutibilidade é particularmente útil em contextos acadêmicos e corporativos, facilitando a colaboração entre equipes.

O sistema também demonstrou potencial de escalabilidade, uma vez que novos scrapers podem ser integrados sem alterações profundas na base de código. Essa característica o torna aplicável não apenas ao mercado imobiliário, mas também a outros domínios que exigem coleta e análise de grandes volumes de dados não estruturados.

6. Discussão

A análise dos resultados evidencia que a arquitetura modular e distribuída adotada neste trabalho oferece vantagens significativas em termos de flexibilidade e manutenção. A separação entre scraping, normalização e disponibilização favorece o isolamento de falhas e facilita a substituição de componentes individuais sem impacto global.

Apesar dos resultados positivos, algumas limitações foram identificadas. A principal delas é a ausência de um mecanismo automático de deduplicação de imóveis, que permitiria identificar listagens repetidas entre diferentes portais. A implementação de algoritmos baseados em similaridade textual e geográfica poderia melhorar a qualidade dos dados armazenados.

Outra limitação é a necessidade de manutenção constante dos scrapers, já que pequenas alterações no layout dos sites podem comprometer a coleta. Nesse sentido, a criação de testes automatizados de scraping e sistemas de monitoramento de falhas se mostra essencial para aumentar a robustez da solução.

No âmbito da infraestrutura, a conteinerização garantiu reprodutibilidade, mas a escalabilidade horizontal ainda depende da adoção de ferramentas de orquestração, como Kubernetes ou Docker Swarm. Tais ferramentas permitiriam o balanceamento dinâmico de carga e a execução paralela de múltiplos scrapers.

Além disso, o uso de autenticação JWT provou-se eficaz, mas futuras versões podem incluir autorização baseada em papéis (RBAC) e políticas de auditoria, aumentando o controle de acesso e a rastreabilidade de ações.

7. Conclusões

O presente trabalho apresentou o desenvolvimento e validação de um sistema de web scraping voltado ao mercado imobiliário, integrando coleta automatizada, normalização de dados e disponibilização via API RESTful e interface web interativa.

A combinação das tecnologias Ruby on Rails e Vue.js mostrou-se eficaz para a construção de uma solução escalável e modular. A integração de tarefas assíncronas e conteinerização Docker elevou a confiabilidade e a reprodutibilidade do sistema, demonstrando que é possível criar uma ferramenta robusta e de fácil implantação.

Os resultados obtidos comprovam que o sistema atende à necessidade de reduzir o esforço manual na coleta e atualização de informações e fornece uma base sólida para análises de mercado, visualizações e integração com outros sistemas.

Como perspectivas futuras, destacam-se:

- o desenvolvimento de mecanismos automáticos de deduplicação e verificação de consistência,
- a integração de ferramentas de análise e visualização de dados,
- a adoção de pipelines de integração e deploy contínuo, e
- o uso de técnicas de aprendizado de máquina para prever variações de preço e comportamento de mercado.

Com essas evoluções, o sistema poderá atuar não apenas como um agregador de dados, mas como uma plataforma analítica completa, contribuindo para pesquisas científicas, tomadas de decisão corporativas e políticas de planejamento urbano baseadas em dados.

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/.