

Implementando Arquitetura Limpa em uma API REST Laravel baseada em MVC: Da estrutura MVC a um design orientado a camadas independentes*

William Marrion Costa da Trindade¹, Leandro Oliveira Freitas¹

¹Universidade Federal de Santa Maria (UFSM)
Santa Maria – RS – Brasil

`william.trindade@acad.ufsm.br`

Abstract. *This paper aims to analyze the refactoring process of a REST API, migrating from Laravel's native MVC structure to Clean Architecture. The study evaluates the impact of decoupling business rules, contrasting the increase in structural complexity with the gains in testability and maintainability. The methodology consisted of the incremental reconstruction of a base API, isolating the domain through Entities and Use Cases, and implementing dependency inversion via Boundaries and Repositories. The comparative analysis highlights the transition from a flow coupled to the Eloquent ORM to a framework-agnostic design. The results indicate that, although Clean Architecture imposes a steeper learning curve and greater verbosity, it is technically viable in PHP and effective in ensuring software longevity, repositioning MVC as a peripheral delivery mechanism rather than the application's core.*

Resumo. *Este artigo tem como objetivo analisar o processo de refatoração de uma API REST, migrando da estrutura MVC nativa do Laravel para a Arquitetura Limpa. O estudo avalia o impacto do desacoplamento das regras de negócio, contrapondo o aumento da complexidade estrutural aos ganhos em testabilidade e manutenibilidade. A metodologia consistiu na reconstrução incremental de uma API base, isolando o domínio através de Entidades e Casos de Uso, e implementando a inversão de dependência através de Boundaries e Repositórios. A análise comparativa evidencia a transição de um fluxo acoplado ao Eloquent ORM para um design agnóstico ao framework. Os resultados indicam que, embora a Arquitetura Limpa imponha uma curva de aprendizado mais íngreme e maior verbosidade, ela é tecnicamente viável em PHP e eficaz para garantir a longevidade do software, reposicionando o MVC como um mecanismo de entrega periférico e não como o núcleo da aplicação.*

1. Introdução

A busca pela qualidade no desenvolvimento de software moderno exige a criação de sistemas que sejam, simultaneamente, robustos e flexíveis [Pressman and Maxim 2021]. Para lidar com a crescente complexidade das aplicações web, o padrão *Model-View-Controller* (MVC) estabeleceu-se como a arquitetura dominante em *frameworks* populares, a

*Este artigo é derivado do Trabalho de Conclusão de Curso (TCC) do autor, orientado pelo Prof. Dr. Leandro Oliveira Freitas, defendido no curso de Sistemas para Internet da UFSM em 2025.

exemplo do Laravel, um dos principais expoentes da linguagem de programação PHP [Grove and Ozkan 2011, LARAVEL 2025]. Embora o MVC promova uma organização inicial eficaz, sua aplicação padrão tende a gerar acoplamento entre regras de negócio e infraestrutura à medida que o software escala, dificultando a manutenção e a testabilidade a longo prazo.

Em contrapartida, a Arquitetura Limpa [Martin 2019] propõe um modelo onde o núcleo da aplicação (regras de negócio) permanece agnóstico a *frameworks* e bancos de dados, protegido pela Regra da Dependência. No entanto, existe uma lacuna significativa na literatura técnica no que tange à documentação prática do processo de refatoração de sistemas pré-existentes baseados no padrão MVC-Web, especialmente no ecossistema PHP com Laravel. Embora a aplicação da Arquitetura Limpa seja discutida em outros cenários de desenvolvimento, a escassez de guias detalhados sobre essa transição específica motiva a realização deste estudo.

Este artigo aborda essa lacuna ao apresentar os resultados obtidos no Trabalho de Conclusão de Curso (TCC) do autor, onde foi desenvolvida e validada uma estratégia de refatoração para APIs REST. Diferente de abordagens puramente teóricas, o TCC focou na engenharia da solução, detalhando como desmontar o acoplamento nativo do *framework* para implementar os conceitos da Arquitetura Limpa de maneira funcional.

O objetivo principal foi demonstrar a viabilidade técnica e os impactos arquiteturais dessa migração. O estudo partiu de uma aplicação baseada no padrão MVC e executou uma reestruturação completa em camadas independentes (Entidades, Casos de Uso, Adaptadores de Interface e *Frameworks e Drivers*).

Nesse sentido, o presente artigo contribui com uma análise dos resultados obtidos no TCC, bem como fornece um guia teórico sobre a organização e as etapas necessárias para a refatoração de APIs REST em Laravel, partindo do MVC para a Arquitetura Limpa.

2. Fundamentação Teórica

Esta seção apresenta os conceitos arquiteturais que fundamentam este estudo, partindo do padrão MVC tradicional, passando pelos princípios de design SOLID e culminando na Arquitetura Limpa.

2.1. O Padrão MVC e suas Adaptações Web

O padrão *Model-View-Controller* (MVC) foi originalmente concebido por Trygve Reenskaug no final da década de 1970 para aplicações *desktop* em Smalltalk, com o objetivo de separar a representação da informação da interação do usuário [Reenskaug 1979]. A arquitetura divide a aplicação em três componentes principais: o *Model*, responsável pela lógica de negócios e persistência; a *View*, encarregada da apresentação; e o *Controller*, que gerencia a entrada de dados [Reenskaug 2003].

No contexto do desenvolvimento web moderno, *frameworks* como o Laravel adotaram uma variação deste padrão, frequentemente denominada MVC-Web [Grove and Ozkan 2011]. Diferente do modelo original, onde a *View* observava o *Model* diretamente, no MVC-Web o *Controller* assume um papel central de orquestração, interceptando a requisição HTTP, manipulando o *Model* e selecionando a *View* para a resposta. Embora eficaz para o desenvolvimento rápido, essa centralização frequentemente

leva à criação de “Controllers Gordos” (*Fat Controllers*) e ao acoplamento forte entre as regras de negócio e a camada de persistência (Active Record/Eloquent), dificultando a testabilidade e a manutenção evolutiva [Stauffer 2023].

2.2. Princípios SOLID e Inversão de Dependência

Para mitigar os problemas de acoplamento, a aplicação de princípios de design orientado a objetos torna-se essencial. Os princípios SOLID, compilados por [Martin 2019], fornecem diretrizes para a construção de *software* flexível. Dentre eles, dois são cruciais para a refatoração proposta neste trabalho:

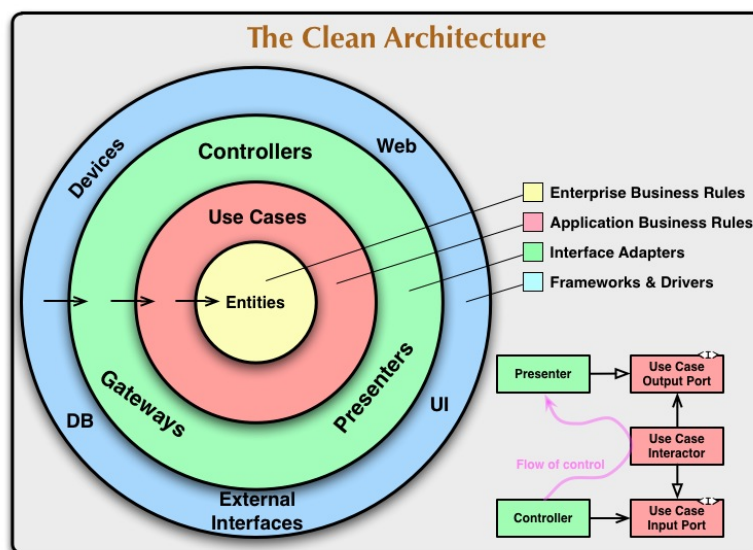
- **Princípio da Responsabilidade Única (SRP):** Estabelece que uma classe deve ter apenas uma razão para mudar. A violação deste princípio é comum em *Controllers* que acumulam validação, lógica de negócio e resposta HTTP [Ramachandrappa 2024].
- **Princípio da Inversão de Dependência (DIP):** Preconiza que módulos de alto nível não devem depender de módulos de baixo nível; ambos devem depender de abstrações. Isso permite que o núcleo da aplicação não dependa de detalhes como o banco de dados ou o *framework web* [Joshi 2016].

2.3. Arquitetura Limpa

A Arquitetura Limpa (*Clean Architecture*), proposta por [Martin 2019], integra conceitos como Arquitetura Hexagonal e *Onion Architecture* em um modelo unificado. Seu objetivo é criar sistemas independentes de *frameworks*, testáveis e independentes de interface gráfica ou banco de dados.

A Figura 1 apresenta o diagrama de referência da Arquitetura Limpa proposto por [Martin 2019]. Esta representação visual organiza o sistema em camadas concêntricas, onde cada círculo delimita um nível específico de responsabilidade e abstração, fundamentando-se na Regra da Dependência para isolar as regras de negócio dos detalhes de infraestrutura.

Figura 1. A arquitetura limpa.



Fonte: [Martin 2019].

As camadas da Arquitetura Limpa são definidas como:

1. **Entidades:** O núcleo central que encapsula as regras de negócio corporativas críticas e independentes da aplicação.
2. **Casos de Uso:** A camada que orquestra o fluxo de dados para e a partir das entidades, contendo as regras de negócio específicas da aplicação.
3. **Adaptadores de Interface:** Camada responsável por converter dados do formato mais conveniente para os casos de uso e entidades para o formato conveniente para agentes externos (Web, DB). Aqui residem os Controllers e Presenters.
4. **Frameworks e Drivers:** A camada mais externa, composta por detalhes técnicos como o *framework web* (Laravel), banco de dados e interfaces de usuário.

Essa estrutura promove o conceito de “Arquitetura Gritante” (*Screaming Architecture*), onde a organização do projeto comunica seu propósito de negócio (ex: “Sistema de Contatos”) e não a ferramenta utilizada para construí-lo [Martin 2019].

2.4. Trabalhos Relacionados

A adoção da Arquitetura Limpa tem sido amplamente explorada na literatura recente, abrangendo desde aplicações móveis até ecossistemas complexos de microsserviços. No cenário de desenvolvimento móvel (*mobile*), [BUENO 2021] e [Pinto et al. 2021] demonstram a viabilidade da aplicação destes princípios utilizando o *framework* Flutter. Enquanto [BUENO 2021] foca na estruturação arquitetural geral de aplicativos para isolamento da interface de usuário, [Pinto et al. 2021] detalham essa implementação no contexto de um aplicativo diário de enxaqueca, evidenciando como a separação rigorosa de responsabilidades facilita a testabilidade tanto da interface quanto das regras de negócio em ambientes móveis.

No domínio de sistemas voltados para a área da saúde, [Souza 2021] descreve o desenvolvimento de um *software* para indicações médicas. O autor destaca como a Arquitetura Limpa atua na proteção de regras de negócio corporativas sensíveis, isolando-as de eventuais mudanças nas interfaces de usuário ou nos mecanismos de persistência de dados.

No contexto de sistemas distribuídos e aplicações *backend*, [P and H 2023] propõem uma abordagem que integra a Arquitetura Limpa à construção de microsserviços utilizando a linguagem Python, com foco em alta escalabilidade. De maneira complementar, [Boff et al. 2023] aplicam a mesma arquitetura no desenvolvimento de um Serviço de Gerenciamento de Identidades e Acessos (IAM), demonstrando a eficácia do padrão estrutural para manter o isolamento de domínios complexos em arquiteturas orientadas a serviços.

Apesar das contribuições relevantes desses estudos, observa-se que a literatura atual concentra-se majoritariamente na criação de novos sistemas, explorando predominantemente ecossistemas móveis ou arquiteturas distribuídas baseadas em Python e Java. Este artigo, por sua vez, diferencia-se ao preencher a lacuna metodológica referente à refatoração de uma API REST pré-existente baseada no padrão MVC-Web tradicional, especificamente no ecossistema PHP com Laravel.

Ao contrário de uma implementação concebida a partir do zero, conforme abordado pelas literaturas correlatas, o presente estudo contribui ao documentar o esforço

estrutural necessário para desmembrar o forte acoplamento nativo do padrão MVC, utilizado nos principais *frameworks* de desenvolvimento *web*, oferecendo um guia analítico e comparativo direto entre a agilidade inicial do MVC e o design orientado a camadas independentes que a Arquitetura Limpa propõe.

3. Metodologia

A metodologia adotada no TCC configurou-se como um estudo de caso prático de natureza aplicada. O objeto de estudo foi uma API REST de gerenciamento de contatos, inicialmente desenvolvida seguindo o padrão MVC nativo do *framework* Laravel, baseada no material didático produzido pelo Programa de Educação Tutorial (PET) do curso de Sistemas de Informação da UFSM¹.

Para garantir a reprodutibilidade do experimento, o ambiente de desenvolvimento foi padronizado utilizando a tecnologia de *containers* Docker. A infraestrutura foi definida através do Docker Compose, orquestrando *containers* isolados para a aplicação (PHP 8.2), servidor web (Nginx) e banco de dados (MySQL 8).

O procedimento experimental foi dividido em três etapas sequenciais:

1. **Implementação da Linha de Base (MVC):** Codificação da API seguindo estritamente as convenções do Laravel (Controllers acoplados ao Eloquent ORM), servindo como grupo de controle para comparação.
2. **Refatoração para Arquitetura Limpa:** Reestruturação incremental do código-fonte aplicando as camadas concêntricas propostas por [Martin 2019]. Esta etapa focou na extração de lógica de negócio dos Controllers para *Interactors* (Casos de Uso) e no isolamento do domínio através de Entidades puras e Interfaces.
3. **Análise Comparativa:** Exame das diferenças arquiteturais entre o MVC e a Arquitetura Limpa, com foco na decomposição de responsabilidades, na aplicação da Regra da Dependência no fluxo de dados e nos impactos observados sobre a manutenibilidade do sistema.

Este método permitiu isolar as variáveis arquiteturais e observar diretamente os impactos da aplicação dessa arquitetura em APIs REST que usam MVC no Laravel.

4. Desenvolvimento

O desenvolvimento deste estudo de caso foi estruturado de forma incremental, iniciando pela construção de uma API REST baseada no padrão MVC nativo do *framework* Laravel para, subsequentemente, aplicar a refatoração orientada aos princípios da Arquitetura Limpa.

Para fins de concisão e profundidade técnica, este artigo detalha o processo de refatoração focado no caso de uso “Criar Contato” (*CreateContact*), que ilustra o fluxo completo da arquitetura. A implementação integral da API, abrangendo todas as operações CRUD (Create, Read, Update, Delete), encontra-se disponível no repositório público do projeto².

¹Disponível em: <https://www.ufsm.br/pet/sistemas-de-informacao/2020/08/25/api-laravel>

²Disponível em: <https://github.com/williamtrindade/clean-architecture-laravel-api>

4.1. Implementação da Arquitetura Limpa

Após a recriação da API base desenvolvida pelo grupo PET, a refatoração seguiu uma abordagem incremental, iniciando pelo desenvolvimento do núcleo da arquitetura em direção às camadas externas.

4.1.1. Camada de Entidades

A primeira etapa consistiu no isolamento das regras de negócio corporativas. A classe `Contact` foi reescrita como uma entidade pura (Plain Old PHP Object), desvinculada da classe `Model` do Eloquent. Nesta camada, foram implementadas as validações intrínsecas de domínio (como formato de e-mail e telefone) diretamente nos métodos modificadores, garantindo a integridade dos dados independentemente da persistência.

4.1.2. Camada de Casos de Uso

Nesta camada, as operações do sistema foram segregadas em classes de Caso de Uso (*Interactors*). Para garantir o desacoplamento, foram definidos os seguintes componentes:

- **Contracts:** Criação da interface `ContactRepositoryInterface`, abstraindo o acesso aos dados.
- **DTOs (Input/Output Data):** Definição de classes imutáveis como `CreateContactRequestModel` e `CreateContactResponseModel` para transporte de dados entre camadas.
- **Boundaries:** Interfaces de entrada (*InputBoundary*) e saída (*OutputBoundary*) para estabelecer os contratos de comunicação do Interactor.
- **Exceptions:** Implementação de exceções semânticas como `BusinessRuleException` para tratamento de erros de domínio.

4.1.3. Camada de Adaptadores de Interface

A terceira etapa focou na adaptação dos dados entre o formato da web/banco e o formato dos casos de uso. Os **Controllers** foram refatorados para o padrão *Single Action*, atuando apenas como conversores da requisição HTTP para os DTOs de entrada. Para a persistência, foi implementado o **Repositório Concreto** (`ContactRepository`). Este componente atua como um adaptador que converte a Entidade pura para o Eloquent `Model` e vice-versa, utilizando um método de mapeamento (*Mapper*) explícito. Para a resposta, foram criados **Presenters** (ex: `CreateContactApiPresenter`) e **View-Models**, preparando os dados que saem da camada mais interna para serem apresentados na camada mais externa da API.

4.1.4. Camada de Frameworks e Drivers

A fase final consistiu na organização dos detalhes técnicos. Os `Models` do Eloquent foram movidos para um diretório específico de infraestrutura. Para conectar as interfaces abstratas às implementações concretas, foi criado o `CleanArchServiceProvider`.

Este provedor utiliza o container de injeção de dependência do Laravel para realizar os vínculos (*bindings*) entre o Repositório, os Interactors e suas respectivas interfaces. Por fim, configurou-se o tratamento global de exceções no arquivo `bootstrap/app.php`, convertendo as exceções de domínio em respostas HTTP padronizadas (ex: 422 Unprocessable Entity), mantendo os controladores limpos de blocos *try-catch*.

5. Resultados e Discussão

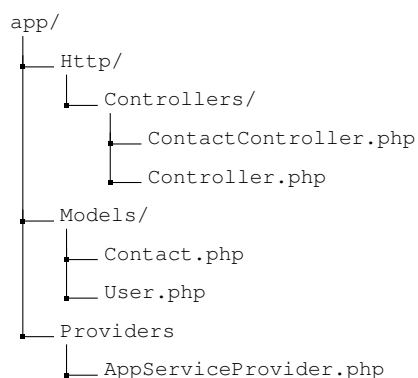
Para validar a eficácia da refatoração proposta, os resultados são apresentados através de uma análise comparativa direta entre a API base (MVC) e a versão refatorada (Arquitetura Limpa). A avaliação divide-se em três eixos: a reestruturação dos componentes, a alteração no fluxo de controle e os resultados gerais observados em termos de engenharia de software.

5.1. Comparação Estrutural

A mudança mais imediata foi a transição de uma organização orientada a *framework* para uma organização orientada ao domínio (“Arquitetura Gritante”).

No modelo MVC original (Figura 2), a estrutura de diretórios agrupava arquivos por sua natureza técnica (ex: `Http/Controllers`, `Models`), ocultando as regras de negócio.

Figura 2. Estrutura - MVC



Fonte: Autor.

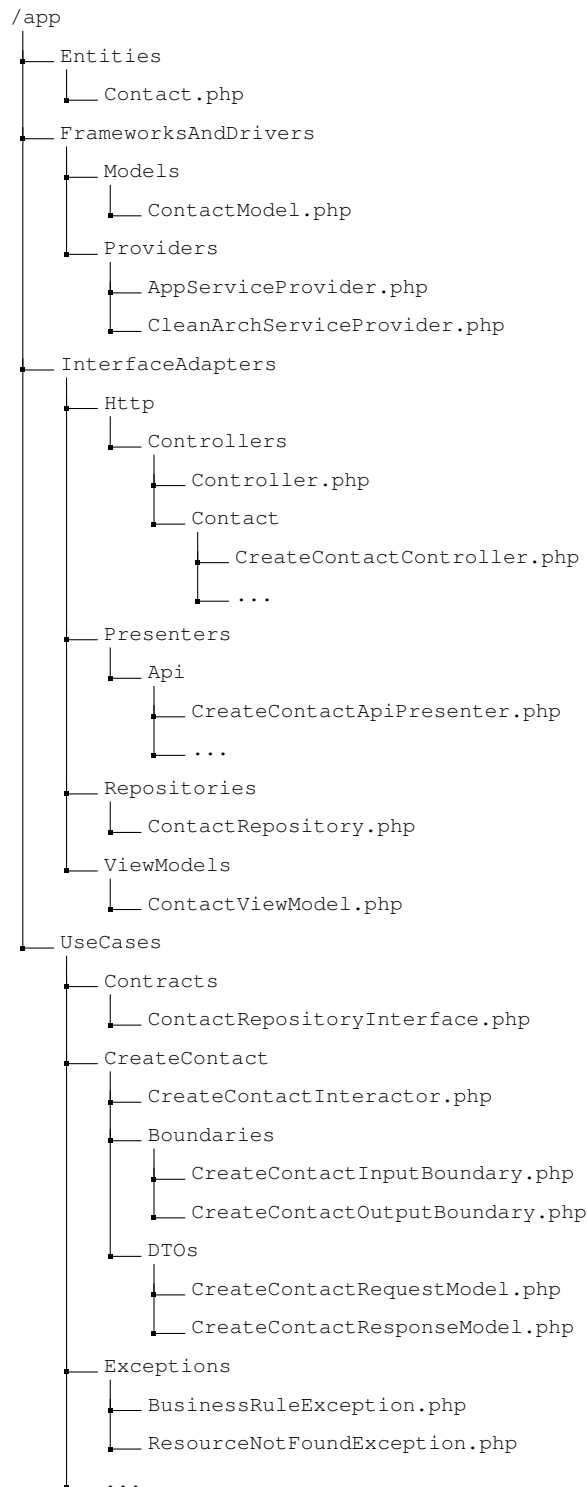
Já na versão refatorada (Figura 3), a estrutura passou a comunicar as funcionalidades do sistema. O diretório `app` foi reorganizado em camadas concêntricas: *Entities* (Núcleo), *UseCases* (Regras de Aplicação), *InterfaceAdapters* (Adaptadores) e *FrameworksAndDrivers* (Infraestrutura).

Essa segregação estrutural permitiu isolar completamente o código de negócio (agora em *UseCases*) das dependências do Laravel, que foram movidas para as camadas mais externas (*FrameworksAndDrivers*), facilitando a navegação e a compreensão do propósito do sistema.

5.2. Comparação do Fluxo de Requisição

A refatoração alterou radicalmente o comportamento interno da aplicação, substituindo o acoplamento direto pela inversão de dependência.

Figura 3. Estrutura - Arquitetura Limpa

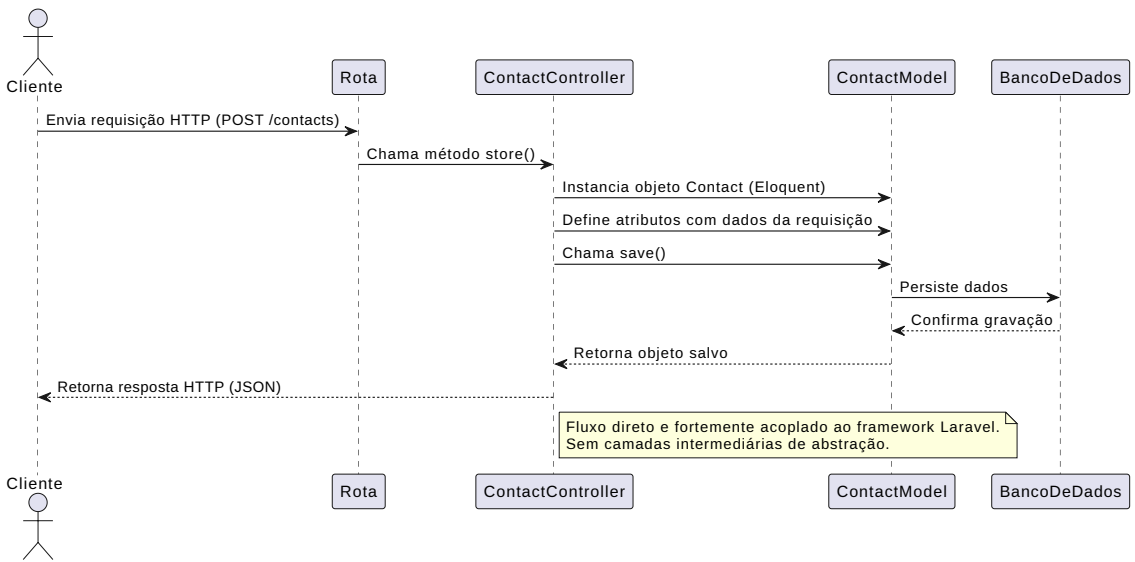


Fonte: Autor.

A Figura 4 ilustra o fluxo de controle linear da implementação original baseada no padrão MVC. Nesta abordagem, observa-se um acoplamento direto onde o *Controller* interage diretamente com o *Model* (Eloquent). Como o *Model* no Laravel herda funcionalidades de persistência ativa (*Active Record*), as regras de negócio acabam misturadas com

a lógica de banco de dados, tornando o fluxo dependente da infraestrutura do framework.

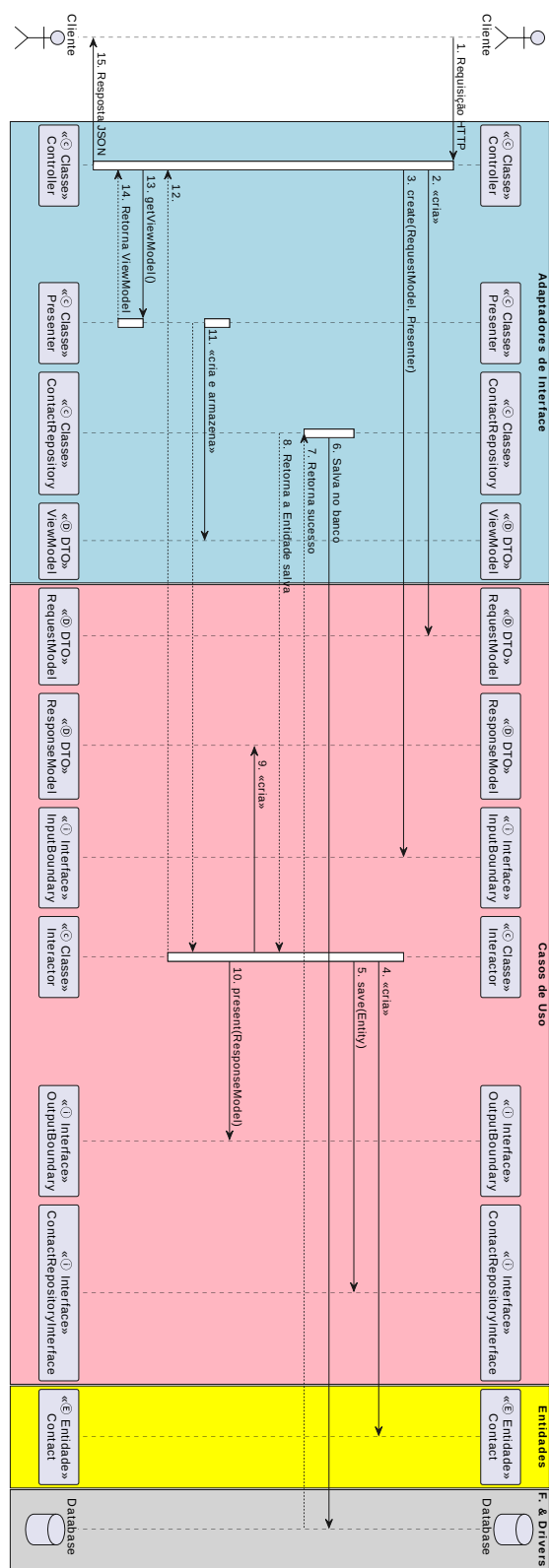
Figura 4. Fluxo da Requisição - MVC.



Fonte: Autor.

Em contrapartida, a Figura 5 detalha o fluxo refatorado, evidenciando a inversão de dependência. O controle inicia no *Controller*, mas só atinge o núcleo da aplicação através de interfaces (*Input Boundary*), garantindo que a camada externa não conheça a implementação interna. O *Interactor* processa a regra de negócio e acessa os dados através de outra interface (*ContactRepositoryInterface*), desconhecendo completamente a existência do banco de dados ou do Eloquent, o que isola o domínio de detalhes externos.

Figura 5. Fluxo da Requisição - Arquitetura Limpa.



Fonte: Autor.

5.3. Resultados Gerais e Impactos

A análise qualitativa dos resultados, conforme discutido no desenvolvimento do trabalho, revelou um *trade-off* claro entre complexidade inicial e manutenibilidade a longo prazo:

- **Curva de Aprendizado e Cultura da Equipe:** A adoção da Arquitetura Limpa impõe uma sobrecarga cognitiva inicial significativa. Diferente do desenvolvimento rápido propiciado pelo Laravel padrão, esta abordagem exige que a equipe de desenvolvimento possua conhecimentos sólidos em orientação a objetos, injeção de dependência e princípios SOLID. Para equipes formadas majoritariamente por desenvolvedores júnior, a abstração excessiva pode reduzir a produtividade inicial até que o modelo mental seja absorvido.
- **Complexidade Estrutural vs. Escopo do Projeto:** Observou-se que a arquitetura introduz uma verbosidade considerável (aumento no número de arquivos e camadas) que se configura como um “excesso de engenharia” para projetos de escopo reduzido, MVPs ou simples CRUDs. A Arquitetura Limpa justifica-se, portanto, em aplicações de longa vida útil e alta complexidade de regras de negócio, onde a organização supera o custo da burocracia inicial.
- **Qualidade de Software e Testabilidade:** O ganho mais expressivo foi a garantia de qualidade através de testes. O desacoplamento demonstrou a viabilidade de escrita de testes unitários rápidos e isolados para os Casos de Uso, sem a necessidade de carregar o framework ou o banco de dados. Isso mitiga a fragilidade comum em testes de integração e fornece um *feedback* imediato para a equipe durante o desenvolvimento de novas funcionalidades.
- **Independência e Longevidade:** O estudo validou a transformação do framework Laravel em um “detalhe” de implementação. Ao isolar o domínio, o sistema tornou-se resiliente a mudanças tecnológicas externas (como a troca de bibliotecas ou drivers de banco de dados). Isso assegura que o núcleo da aplicação permaneça imutável frente à obsolescência tecnológica, protegendo o investimento intelectual da empresa a longo prazo.

6. Conclusão

O estudo demonstrou a viabilidade técnica e os impactos arquiteturais da aplicação da Arquitetura Limpa em projetos desenvolvidos com o framework Laravel. Através da refatoração de uma API REST baseada no padrão MVC, foi possível isolar as regras de negócio de detalhes de infraestrutura, invertendo o fluxo de dependência e promovendo um desacoplamento efetivo.

Os resultados obtidos confirmam que a adoção da Arquitetura Limpa exige um investimento inicial significativo em termos de verbosidade e complexidade estrutural. A criação de camadas intermediárias (Casos de Uso, DTOs, Presenters) aumentou a quantidade de artefatos de código em comparação com a simplicidade do MVC nativo do Laravel. No entanto, esse custo traduziu-se diretamente em benefícios de longo prazo: a facilitação na escrita de testes automatizados com *mocks*, a independência do mecanismo de persistência e a clareza semântica do domínio (“Arquitetura Gritante”).

Conclui-se que, para protótipos rápidos ou sistemas de baixa complexidade, o padrão MVC do Laravel permanece superior devido à sua agilidade de desenvolvimento. Entretanto, para aplicações corporativas de longa duração, onde a manutenibilidade e

a qualidade de software são críticas, a migração para a Arquitetura Limpa representa uma decisão estratégica válida, transformando o framework de uma restrição arquitetural central em um detalhe plugável.

Apesar dos benefícios qualitativos observados, este estudo limitou-se à refatoração de uma única aplicação base, o que restringe a generalização ampla dos resultados. Assim, como trabalhos futuros, propõe-se a aplicação e o teste deste modelo de refatoração em outras amostras de sistemas, especialmente em aplicações legadas e de maior complexidade. Além disso, a realização de uma análise quantitativa de desempenho é de suma importância e necessária, mas não foi abordada neste escopo. Portanto, sugere-se também a criação de um *benchmark* estruturado, detalhando as especificações de *hardware*, conexões de rede e banco de dados para monitorar o fluxo de requisições e avaliar comparativamente o custo operacional e o tempo de resposta entre o padrão MVC e a Arquitetura Limpa.

Referências

- Boff, G. d. S. et al. (2023). Serviço para gerenciamento de identidades e acessos baseado em arquitetura limpa e microsserviços.
- BUENO, C. E. D. O. (2021). Desenvolvimento de um aplicativo utilizando o framework flutter e arquitetura limpa.
- Grove, R. F. and Ozkan, E. (2011). The mvc-web design pattern. In *International Conference on Web Information Systems and Technologies*, volume 2, pages 127–130. SCITEPRESS.
- Joshi, B. (2016). *Beginning SOLID principles and design patterns for ASP. NET developers*. Springer.
- LARAVEL (2025). Laravel - The PHP Framework For Web Artisans. <https://laravel.com/docs>. Acessado em: 26 de outubro de 2025.
- Martin, R. C. (2019). *Arquitetura Limpa: O guia do artesão para estrutura e design de software*. Alta Books Editora.
- P, I. V. and H, V. P. (2023). An approach to clean architecture for microservices using python. In *2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*, pages 1–5.
- Pinto, R. S. et al. (2021). Um aplicativo para registro de diários de enxaqueca utilizando flutter e os princípios da arquitetura limpa.
- Pressman, R. S. and Maxim, B. R. (2021). *Engenharia de software-9*. McGraw Hill Brasil.
- Ramachandrapa, N. C. (2024). Solid design principles in software engineering. *International Journal of Computer Trends and Technology*, 72(9):18–23.
- Reenskaug, T. (2003). The model-view-controller (mvc) its past and present trygve reenskaug, university of oslo (trygver@ifi.uio.no).
- Reenskaug, T. M. H. (1979). The original mvc reports.
- Souza, G. (2021). Desenvolvimento de software para indicação médica utilizando arquitetura limpa.

Stauffer, M. (2023). *Laravel: Up et running: A framework for building modern Php Apps*.
O'Reilly Media, 3rd edition edition.