Uma Proposta de Arquitetura de Software Limpa baseada em Microsserviços

Francisco C. Dantas, Dr. Leonardo A. Casillo, Dr. Francisco Milton M. Neto

Departamento de Computação - Universidade Federal Rural do Semi-Árido
Caixa Postal BR 110 - Km 47 Bairro Pres. Costa e Silva, Campus Universitário – 59625-900 – Mossoró – RN – Brasil

{Francisco.dantas@ufersa.edu.br, casillo@ufersa.edu.br, miltonmendes@ufersa.edu.br

Abstract. *This article presents a study focused on the description and proposal of a clean software architecture based on microservices. We look for a more pragmatic specification of clean software architecture based on microservices, aiming at the open software lifecycle. The options regarding the chosen technologies will be used project standards in the development, deployment, operation and maintenance cycle that should mininize the cost and maxinize the system's service life giving more productivity to those involved.*

Resumo. *Este artigo apresenta um estudo voltado à descrição e proposta de uma arquitetura de software limpa baseada em microsserviços. Buscamos em uma especificação mais pragmática de arquitetura de software limpa baseada em microsserviços, com objetivo no ciclo de vida de software aberto. As opções em relação às escolhas de tecnologias adotadas, serão utilizados padrões de projetos no ciclo de desenvolvimento, implantação, operação e manutenção que devem mininizar o custo e maxinizar a vida útil do sistema dando mais produtividade aos envolvidos.*

1. Introdução

Em meados da década de 60, o mercado de tecnologia sofreu com a crise do *software*. Durante esse período crítico muitos problemas relacionados ao prazo, custos de projeto de *software* causaram uma baixa produtividade, baixa qualidade e difícil manutenção de aplicações. Ao longo dos anos 70, houve o surgimento de estudos e pesquisas com o objetivo de criar e melhorar os processos de desenvolvimento. Com o avanço nos estudos na área de engenharia e arquitetura de *software*, foram desenvolvidas ferramentas, métodos, padrões e técnicas para auxiliar no processo de desenvolvimento de *software* de qualidade.

Segundo Sommerville (2011, p. 103): "O projeto de arquitetura está preocupado com a compreensão de como um sistema deve ser organizado e com sua estrutura geral desse sistema." Em torno de uma visão sobre a arquitetura de *software*, algumas decisões de projeto, seja na estrutura ou de software devem ser realizadas no contexto de requisitos, modelos de processos, padrões de projetos, aspectos comportamentais e não apenas em tendências de tecnologias de mercado.

O problema relacionado ao ciclo de desenvolvimento de programas focado na arquitetura, pode maximizar o esforço e minimizar a produtividade, saber quais atributos da arquitetura de *software* podem mudar para mitigar tais fatores. Será possível manter e reduzir o alto custo da escolha da melhor opção para organização de

desenvolvimento, evitando o excesso de erros, aumentando a qualidade da arquitetura, viabilizando o custo da manutenção de dependências externas?

O Objetivo deste trabalho é propor uma arquitetura de *software* limpa baseada na referência que permita o desenvolvimento modular dos sistemas de software, observadas as necessidades e padrões de projetos. Segundo Martin (2018, p. 247): “A arquitetura de software de um sistema é definida pelos limites especificados dentro desse sistema e pelas dependências que cruzam esses limites.”.

Definiu-se uma visão pragmática na concepção da arquitetura de *software* limpa, centrada no uso de técnicas e ferramentas de validação de qualidade de código como pilares de sustentação. Um protótipo de um software que comprove a flexibilidade, facilitando a compreensão do software em todas as etapas do ciclo de vida, uma análise dos principais aspectos arquitetônicos, concepção e definição adequada aos requisitos propostos com escolhas de tecnologias e mecanismos que permitam a integração de diversos sistemas utilizando tecnologias modernas baseadas em microsserviços.

As tecnologias promovem mudanças gigantescas na sociedade, são revoluções dentro e fora do contexto que estão inseridas, estamos cada vez mais conectados a grande rede de computadores. Segundo (Fowler, 2014,?): “é uma abordagem para desenvolver um único aplicativo como um conjunto de pequenos serviços, cada um executando em seu processo e se comunicando com mecanismos leves, geralmente uma Interface de Programação de Aplicação (API) de recurso HTTP”, [RFC 2616, 1999].

Adotou-se uma metodologia sobre arquiteturas de *software*, padrões de projetos, soluções arquitetônicas em uso pelo mercado de tecnologia. Utilizando metodologia ágil: uma abordagem incremental para a especificação, o desenvolvimento e entrega do software, com adoção do Scrum, segundo (SCHWABER, 2004, SCHWABER e BEEDLE, 2002): “é um método ágil geral, mas seu foco está no gerenciamento do desenvolvimento iterativo, ao invés das abordagens técnicas especificadas da engenharia de software ágil.”. Segundo Kent Beck (2003,?): “O desenvolvimento guiado por teste é uma técnica de desenvolvimento de software que se relaciona com o conceito de verificação e validação e se baseia em um ciclo curto de repetições”.

Ao longo do trabalho foi adotada uma arquitetura de *software* capaz de suportar todo o ciclo de vida do sistema, reduzindo o custo e maximizando a produtividade. Utilizando padrões de projetos, especificação de escopo aberto sobre as tecnologias, bibliotecas, testes de software, documentação de software, foram utilizadas no ciclo de desenvolvimento, alguns padrões de mercado tais como Integração Contínua — CI, Entrega Contínua — CD, todas elas aplicadas ao uso de ferramentas de código aberto, todos os ciclos de entregas são versionados através do sistema de versionamento — Git. O Test Driven Development (TDD) será responsável por manter as responsabilidades de código coeso, funcional, também para solucionar problemas de coesão, dependência e acoplamento entre as camadas da Arquitetura.

2. Arquitetura de software

Imagine uma planta de um prédio. Esse documento foi preparado por um arquiteto, estabelecendo os planos para a construção do projeto. Todas as características com

entradas, saídas, formato de janela, portas e telhado podem ser observados em detalhes do padrão de projeto utilizado. Suponha a arquitetura de uma igreja, as definições de salões, uma sequência de entradas com capacidade para reunir pessoas seria observada de forma distinta, uma igreja possui um padrão de documento arquitetônico e estilos de construção que difere de prédios.

Segundo Sommerville (2011, p.104): "A arquitetura é importante, pois afeta o desempenho e a robustez, bem como a capacidade de distribuição e de manutenção de um sistema. Os requisitos não funcionais dependem da arquitetura do sistema — a forma como esses componentes estão organizados e se comunicam". A compreensão sobre as decisões na visão da arquitetura, usando conhecimentos de padrões genéricos de arquitetura, avaliando-se questões relacionadas ao reuso de componentes na execução do projeto de *software*.

Segundo ANDREW (2007, p.20):

> Há diferentes modos de ver a organização de um sistema distribuído, mas uma maneira óbvia é fazer uma distinção entre a organização lógica do conjunto de componentes de software e, por outro lado, a realização física propriamente dita. A organização de sistemas distribuídos, trata, em grande parte, dos componentes de software que constituem o sistema. Essas arquiteturas de software nos dizem como os vários componentes de software devem ser organizados e como devem interagir.

Devemos acrescentar essa visão abstrata sobre os vários aspectos importantes, necessários para organização dos componentes, representando uma linha de raciocínio, planejada e coesa, sem perder o foco em uma visão conceitual de arquitetura de *software*, auxiliando nas decisões cruciais para o processo de desenvolvimento de sistemas.

Podemos compreender a partir do propósito de uma arquitetura, segundo Martin (2018, p. 197): "As boas arquiteturas devem ser centradas em casos de uso para que os arquitetos possam descrever com segurança as estruturas que suportam esses casos de uso, sem se comprometer com *frameworks*, ferramentas e ambientes".

Ao propor a proposta de boas práticas, seguindo um escopo voltado para arquitetura mais adequada, observamos o quanto uma decisão de tecnologia ou padrões de projetos inadequados podem comprometer uma boa estrutura de projeto de *software*.

3. Estilos arquitetônicos

Segundo ANDREW (2007, p.30): "Tal estilo é formulado em termos de componentes, do modo como esses componentes estão conectados uns aos outros, dos dados trocados entre componente e, por fim, da maneira como esses elementos são configurados em conjunto para formar um sistema."

Os padrões arquitetônicos são uma forma proposta de organização de mais alto nível para sistema de *software*, define relações entre os componentes e módulos principais.

Entre os padrões arquitetônicos:

- Arquitetura em camadas;

- Arquitetura baseada em Microsserviços;
- Arquiteturas orientadas a mensagens;
- Arquiteturas Publish/Subscribe;
- Pipes e Filtros;
- Arquitetura Cliente/Servidor.

Tabela 1. Vantagens e desvantagens de Arquitetura em camadas

Vantagens	Desvantagens
Possibilita trabalhar em níveis crescentes de abstração. Suportam facilmente a aplicação de melhoramentos. Suportam o reuso de componentes de *software*. Facilidade de compreensão, manutenção e desenvolvimento independente.	Alguns sistemas não se adéquam a essa organização. Requisitos de desempenho podem levar a quebra das regras de organização em camadas. Duplicação de funcionalidade, difícil estruturar um sistema através de camadas. Overhead de implementação e desempenho.

4. Arquitetura de Software Limpa

Podemos considerar o conceito mais comum de Arquitetura de *Software* Limpa, segundo Martin (2018, p.12-15):

> Para construir um sistema com um design e uma arquitetura que minimizem o esforço e maximizem a produtividade…. Quanto mais essa arquitetura prefere uma forma à outra, os novos recursos terão mais probabilidade de serem cada vez mais difíceis de encaixar nessa estrutura. Portanto, as arquiteturas devem ser tão agnósticas em sua forma quanto práticas.

O planejamento de uma Arquitetura de *Software* Limpa baseia-se na divisão desse sistema em componentes, na organização desses componentes e nos modos como esses elementos se comunicam entre si, deixando o desenvolvimento modular.

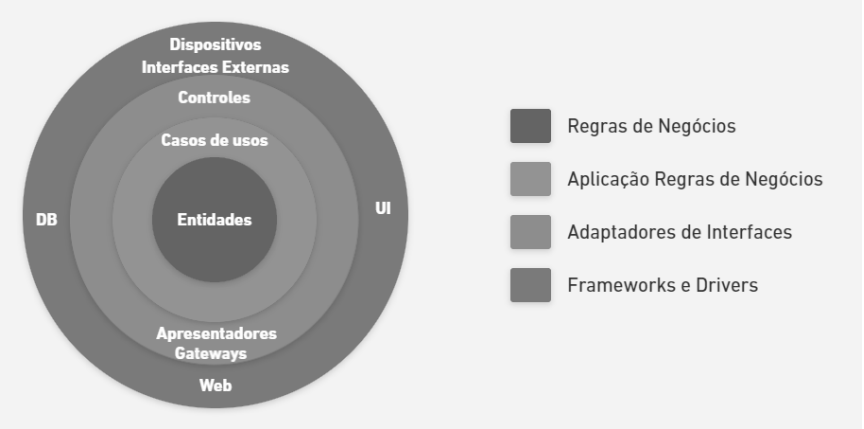

Figura 1. Arquitetura Limpa. Martin (2018, p.203)

Podemos observar como está organizada nossa Arquitetura de *Software* Limpa, vamos comentar um pouco sobre cada camada e detalhes sobre sua importância no projeto de *software*:

- **Entidades:** normalmente são responsáveis por regras de negócio que podem assumir entidades ou casos de usos. Como entidades podemos definir classes comuns para os vários sistemas da empresa, podemos também implementar regras de negócio seguindo de forma genérica.
- **Caso de uso:** podem ser definidas como classes que são menos genéricas, usualmente relativas a um único sistema. Ao implementar um caso de uso podemos ver a regra de negócio que a define, exemplo seria um fluxo de cadastro com dados de uma pessoa física em uma empresa, o usuário terá como pré-requisito possuir um Cadastro de Pessoa Física – CPF para poder solicitar seu registro no sistema.
- **Adaptadores:** possuem a função de converter ou adaptar dados de formatos para outro que possam ser entregues a camada de destino – seguindo o fluxo da regra de dependência.
- **Frameworks externos:** nessa camada todas as implementações de classes pertencem a bibliotecas e frameworks externos ou de terceiros, eles podem definir vários recursos como persistência, construção de interfaces, solicitação de outros serviços como envio de e-mail, comunicação de diferentes hardwares com o *software*. Um ponto muito importante em relação às camadas definidas, quanto mais externas mantemos o fluxo de dependência sobre a camada inferior.

4.1. Princípios SOLID

Uma solução de arquitetura de *software* bem definida desacoplada de dependências de tecnologias que possam impactar a qualidade de *software*, garantindo uma distribuição de responsabilidade entre as camadas da arquitetura, utiliza-se em como um suporte ao desenvolvimento de *software*. Deve-se adotar um bom padrão de desenvolvimento de código utilizando os princípios e padrões como o SOLID que significa:

- S - Princípio de Responsabilidade Única;
- O - Princípio Aberto-Fechado;
- L - Princípio de Substituição de Liskov;
- I - Princípio de Segregação de Interface;
- D - Princípio de Inversão de Dependência.

Com o advento do cenário de constantes atualizações em tecnologias e ferramentas para desenvolvimento de *software*, devemos preparar nosso projeto de arquitetura de *software* para minimizar os custos das mudanças, trocas de bibliotecas ou recursos de banco de dados em aplicações, esperamos no escopo do projeto SmartEye realizarmos tais mudanças de tecnologias ou ferramentas de terceiros reduzindo o alto custo na manutenção de *software*.

A regra de dependência é algo que devemos manter ao implementar todos os níveis da arquitetura de *software*, em uma arquitetura limpa, as classes da camada B não devem conhecer nenhuma classe de uma camada A mais externa.

5. Microsserviços

Segundo Fowler (2014,?): “Um microsserviço é uma coleção de serviços autônomos, que funcionam juntos para realizar operações mais complexas, ele gerencia um conjunto de serviços relativamente simples que podem interagir de maneiras complexas”. Todos os serviços colaboram entre si por protocolos de mensagens independentes de tecnologia, ponto a ponto ou de forma assíncrona. Além de descentralizar as decisões sobre padrões e ferramentas, os microsserviços descentralizam as decisões de armazenamento de dados. Um ponto central de falha que adotado em sistemas monolíticos, cada serviço gerencia seu próprio repositório de dados.

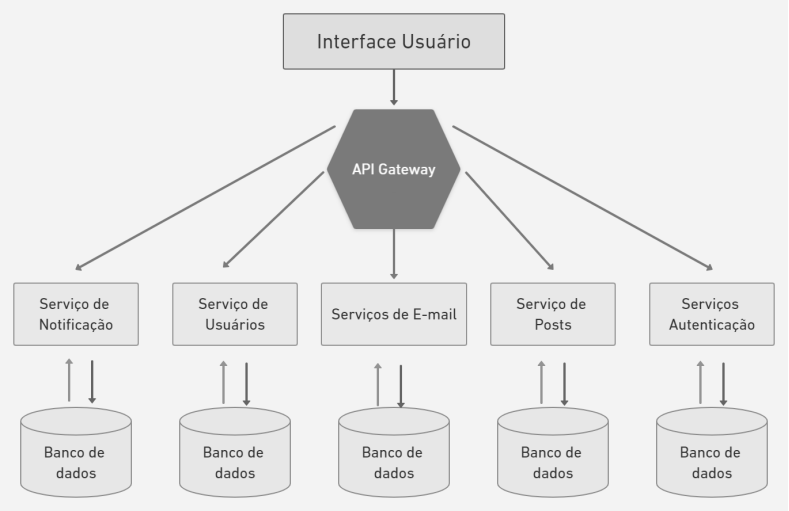

Figura 3. Diagrama arquitetura de microsserviços.

As principais características usadas no projeto SmartEye utilizando os Microsserviços são: alta coesão com acoplamento fraco, uma fonte única de identificação para cada serviço, gerenciamento de tráfego em tempo real, minimizando tabelas de dados para otimizar a carga, execução de monitoramento constante em APIs externas e internas, armazenamento de dados isolado para cada microsserviço, acesso limitado para evitar o acoplamento de serviço, capacidade de dividir a arquitetura monolítica em entidades individuais separadas, escalabilidade, entrega contínua por meio da integração ferramentas de integração contínua.

Tabela 2. Vantagens e desvantagens de Arquitetura Microsserviços

Vantagens	Desvantagens
Pode ser desenvolvido independentemente. Pode ser escalado independentemente. Pode ser reutilizado em múltiplas aplicações. Permite o uso de diferentes linguagens.	Potencialmente muito granulares. Latência durante uso intensivo. Os testes de *software* podem ser complexos.

6. Proposta

Demonstrar a necessidade de padronização arquitetural tanto no âmbito gerencial do desenvolvimento de *software* quanto no âmbito das arquiteturas de *software* convencionais utilizadas em projetos de *software* gerenciando recursos de hardware.

Aplicar a arquitetura de referência no contexto de domínio específico para o desenvolvimento de sistema de *software*, separar o *software* em componentes, do adicionando camadas arquiteturais bem definidas, gerenciar o acoplamento de recursos de hardware, para coleta de dados sobre fluxo de dados utilizando como escopo áreas urbanas utilizando tecnologias da área de Cidades Inteligentes (Smart Cities).

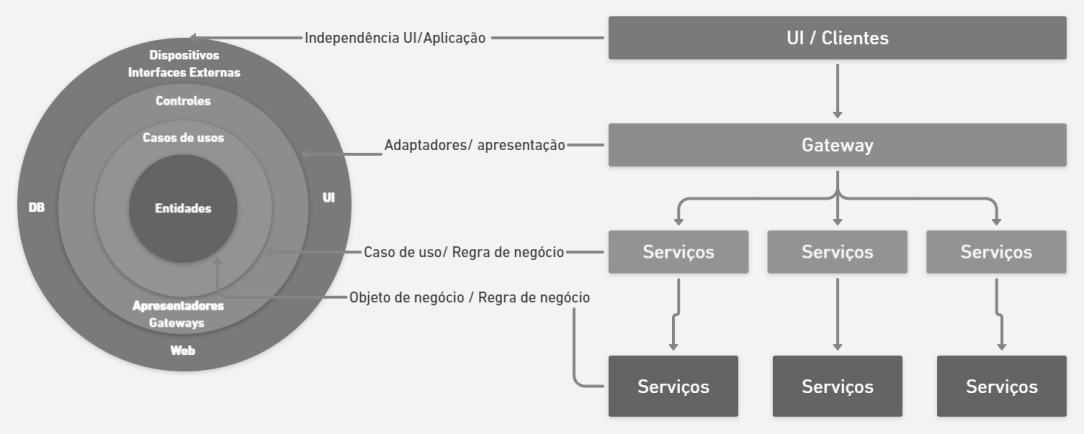

Figura 4. Adaptação arquitetura Limpa. Martin (2018, p.203) e aplicação utilizando microsserviços.

A arquitetura proposta apresentada inspira-se nos princípios da Arquitetura de Software Limpa, Martin (2018, p.229): “Avalie os custos e determine onde estão os limites arquiteturais, e quais deles precisa ser completamente implementado, e quais devem ser parcialmente implementados e quais merecem ser ignorados. ” Uma revisão em pontos de decisões sobre os custos para implementar esses limites pode prejudicar escolhas que deveriam ser ignoradas, isso requer uma boa percepção da arquitetura, suas características e padrões.

7. Projeto SmartEye

O projeto SmartEye é uma solução integrada de hardware, *software* para a identificação de aglomerações e gerenciamento de recursos médicos usando sensores em sistemas embarcados que alimentam bases de dados que utilizam inteligência artificial e o acesso à *interface* de usuário para os gestores de saúde.

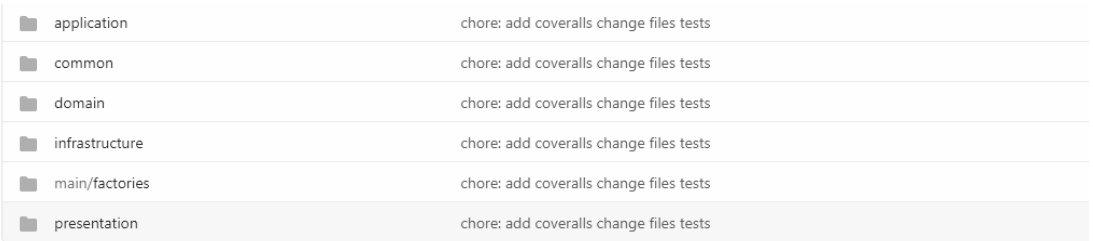

Figura 4. Hierarquia de pastas projeto SmartEye - Back-end.

As estruturas de hierarquia de pastas seguem os contratos firmados na Arquitetura de *Software* Limpa, mantendo os princípios de responsabilidade entre as camadas.

7.1. Sistema de Versionamento Git

Em todo o ciclo de desenvolvimento do projeto de *software* utilizamos uma ferramenta de versionamento de código - Git, com o auxílio de um design de fluxo de trabalho, o Gitflow Workflow, publicado e popularizado pela primeira vez por Vincent Driessen no nvie. O Gitflow define um modelo de ramificação rigoroso projetado com base no lançamento do projeto. Isto oferece uma estrutura robusta para gerenciar projetos maiores.

7.2. Teste de software

Avalia-se como positivo a utilização de boas práticas com o uso da técnica como o TDD para garantir um alcance de qualidade entre todos os membros da equipe de desenvolvimento, no projeto do SmartEye.

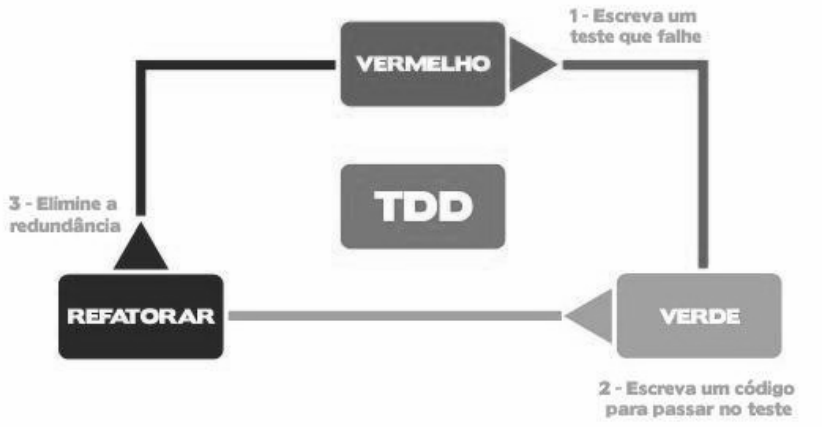

Figura 5. Diagrama desenvolvimento orientado a testes (TDD).

Adotamos como ferramenta de testes para desenvolvimento o Jest, um poderoso *framework* de testes em JavaScript com um foco na simplicidade.

7.3. Entrega e Integração Contínua

O processo de entrega de novos recursos e funcionalidades com frequência aos clientes, pode ser automatizada através de integrações nas etapas do desenvolvimento de aplicações. Os principais conceitos atribuídos a esse modelo passam pela integração, entrega e implantação contínua. Com o CI/CD, podemos solucionar os problemas que as integrações de novos códigos podem causar para as equipes de operações e desenvolvimento.Todo o processo de validação de qualidade de código é realizado no processo de integração da plataforma SmartEye diretamente através de provedores de serviços de *CI/CD*: o Travis-CI e o *Coveralls* cobertura de testes e validação de código.

Figura 6. Relatório de teste de código na plataforma Travis.com.

Como garantia de uma integração que possa ser validada através de processo de teste de *software*, o Travis controla e realiza todos os testes que foram escritos durante o processo de desenvolvimento utilizando o TDD, ele nos permite um controle sobre a entrega de software, baseadas em passar em sua validação final.

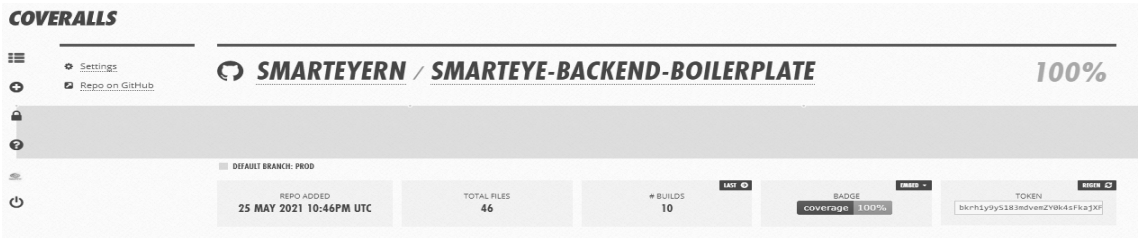

Figura 7.. Cobertura de código portal coveralls.io.

A plataforma de serviço de cobertura de código Coveralls pode nos garantir que todo novo código seja rastreado e isso nos ajude temporalmente para garantir que nosso desenvolvimento esteja testado e validado para novas integrações.

7.4. Documentação ReactJS com Storybook

Segundo Wikipédia (2013): "O React é uma biblioteca JavaScript de código aberto com foco em criar interfaces de usuário (frontend) em páginas web.". O React foi utilizado como biblioteca de componentes *web*, o *Storybook* uma biblioteca de documentação para interface de usuário baseada em *React*, ele agiliza o desenvolvimento, teste e documentação de interface do usuário de forma que possa garantir qualidade de código para todo o ciclo de manutenção, facilitando a troca de informações entre desenvolvedores.

7.5. Documentação de APIs com Swagger

Para a documentação de Interface de Programação de Aplicações (*APIs*), Microsserviços utilizando o *Node.JS*, um *software* baseado no interpretador Javascript - V8 do Google e que permite a execução de códigos fora de um navegador web, utilizaremos o Swagger, uma linguagem de descrição de interface para descrever *APIs* expressas usando *JSON*, um acrônimo de JavaScript Object Notation, o Swagger é usado junto com um conjunto de ferramentas de *software* de código aberto para projetar, construir, documentar e usar serviços da *web*.

8. Análise

As estruturas de hierarquia de pastas seguem os contratos firmados na Arquitetura de Software Limpa, mantendo os princípios de responsabilidade entre as camadas.

O processo de construção da Arquitetura de *Software* Limpa avaliando-se o contexto de Cidades Inteligentes no projeto SmartEye passa pela compreensão de alguns detalhes sobre adaptabilidade e flexibilidade que nossa arquitetura possui como compromisso. Nosso projeto será dividido entre duas grandes estruturas complementares, uma plataforma *web* como cliente (*frontend*), outra denominada como responsável pelas regras de negócio, adotando os microsserviços para disponibilizar diferentes camadas de aplicações para nosso cliente (*backend*)

As estruturas de hierarquia de pastas seguem os contratos firmados na Arquitetura de Software Limpa, mantendo os princípios de responsabilidade entre as camadas. Vamos exemplificar o uso dos princípios adotados na nossa arquitetura de *software* Limpa vamos aos significados do *SOLID* a partir do projeto de Arquitetura de Software Limpa baseada em Microsserviços no projeto do SmartEye.

```
import { Role } from '../role/role';

export interface User {
  id: string;
  email: string;
  first_name: string;
  last_name: string;
  password_hash?: string;
  roles?: Role[];
}
```

Figura 8. Implementação abstração de interface User projeto SmartEye.

Dentro dessa composição limpa seguimos um conjunto de práticas que produzem sistemas independentes de estruturas, testável, independente de interface pré-definida, repositório de dados como bancos relacionais, independentes de bibliotecas ou *frameworks*, essas práticas são detalhes para decisão apenas no momento necessário, deixando nosso contexto pragmático no projeto SmartEye.

```
export const expressRouteAdapter = <T>(controller: Controller<T>) => {
  return async (request: Request, response: Response, next: NextFunction) => {
    return Promise.resolve(
      controller.handleRequest({
        query: request.query,
        params: request.params,
        body: request.body,
        headers: request.headers,
      }),
    )
      .then((controllerResponse) => {
        response
          .status(controllerResponse.statusCode)
          .json(controllerResponse.body);
        return next();
      })
      .catch((error: DefaultApplicationError) => {
        return next(error);
      });
  };
};
```

Figura 9. Implementação padrão de projeto adaptador projeto SmartEye.

Ao analisarmos o projeto, observamos que dentro de cada camada definida na arquitetura de *software* limpa, possuímos um fluxo de dados, as dependências do

código-fonte podem apenas acessar recursos através das camadas inferiores, utilizando a regra de dependência, então não podemos ter acesso de uma camada interna para uma camada mais externa.

9. Considerações finais

Ao definir um projeto de arquitetura de *software* limpa foi proposto que fossem de encontro a uma visão de *software* mais pragmática, não focado em tecnologias e padrões convencionados por recursos externos ao desenvolvimento do *software*, devemos lembrar que devemos suportar todas as etapas de um ciclo de vida do sistema, adotar uma boa arquitetura boa render menos dores de cabeça para todos envolvidos no projeto, clientes satisfeitos com resultados obtidos sem perda de recursos, reduzir o custo da vida útil é sem dúvida primordial, aumentar as entregas dos desenvolvedores é algo muito buscado por corporações.

A definição de uma arquitetura mais amplamente centrada no controle de limites arquitetônicos nos remete a um bom planejamento de código, fatoração de partes mais complexas, aplicação de novos métodos de desenvolvimento e sim troca de tecnologias sem precisar modificar todo o escopo das classes e entidades do projeto.

Dentro deste contexto podemos aplicar um uso de recursos da arquitetura de *software* limpa, mantendo a política de cada camada, cada uma com uma especificação bem definida e com regras de dependências que seguem um fluxo fora para dentro, nenhuma camada poderá viola sem pagar um alto preço, prejudicial ao manter o *software* funcional e com qualidade.

Os microsserviços surgiram como uma proposta para reduzir o acoplamento de aplicações, com implantações independentes, serviços mais especializados e trabalhando em conjunto com uma variedade de linguagens ou tecnologias, ao aplicar a arquitetura de *software* limpa podemos notar uma responsabilidade única, cada serviço separado por camadas bem estruturadas, independente de linguagem ou tecnologia adotada.

9. Agradecimento

À Conselho Nacional de Desenvolvimento Científico e Tecnológico — CNPq pelo apoio financeiro, agradeço a Deus, aos meus pais e a Cinara, minha amiga, esposa e companheira que me ajudou a continuar focado no que eu realmente queria. Agradeço ao Dr. Leonardo Augusto Casillo, meu orientador, pela atenção e auxílio durante a concepção e redação deste trabalho, ao Dr. Francisco Milton Mendes Neto e Victoria Iris Santos de Santana pela contribuição.

Referências

KRUCHTEN, P.; OBBINK, H.; STAFFORD, J. (2006) “The past, present, and future for software architecture". IEEE Software, v. 23, n. 2, p. 22–30.

ANDREW S. TANENBAUM, MAARTEN VAN STEEN. (2007) “Sistemas distribuídos e paradigmas: Princípios e paradigmas". São Paulo: Pearson Prentice Hall.

SOMMERVILLE, I. (2011) “Engenharia de Software”. 9º. ed. São Paulo: Pearson Prentice Hall.

MARTIN, ROBERT. (2018) “Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software”. Rio de Janeiro: Alta Books.

MARTIN, ROBERT. (2012) “The Clean Code Blog: by Robert C. Martin (Uncle Bob) ”. https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html. Junho.

FOWLER, MARTIN. (2019) “Software Architecture Guide. A guide to material on martinfowler.com about software architecture”. Agosto.

DIJKSTRA, EDSGER W. (1972) “The humble programmer”. Commun. ACM 15, 10: 859–866.

SCHWABER, K., BEEDLE, M. (2002) “Agile Software Development with Scrum”. Prentice Hall.

RICHARDSON, CHRIS, (2020) “Refactoring a monolith to microservices”. https://microservices.io/refactoring/index.html. Junho.

WORKING GROUP, NETWORK. (1999) “Hypertext Transfer Protocol - HTTP”. https://datatracker.ietf.org/doc/html/rfc2616. Junho.

REFACTORING.GURU. (2020) “Padrões de Projeto”. https://refactoring.guru/design-patterns/what-is-pattern. Junho.

ZMENDA, ADRIAN. POLAK, ADAM. (2019) “Microservices design patterns for CTOs: API Gateway, Backend for Frontend and more”. https://tsh.io/blog/design-patterns-in-microservices-api-gateway-bff-and-more, setembro.

OLORUNTOBA, SAMUEL. (2020) “SOLID: The First 5 Principles of Object Oriented Design” https://www.digitalocean.com/community/conceptual_articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#single-responsibility-principle, dezembro.

WIKIPÉDIA. (2013). “React (JavaScript) ”. https://pt.wikipedia.org/wiki/React_(JavaScript).