# Introdução

## Finalidade

Este documento possui como objetivo definir os aspectos da Arquitetura do software SUFG e é direcionado aos stakeholders do software a ser desenvolvido, tais como: Gerentes do Projeto, Clientes e equipe técnica, possuindo grande foco para os Desenvolvedores e a Equipe de implantação.

1.2 Escopo

Este documento se baseia no documento de requisitos do SUFG para definir os atributos de qualidade a serem priorizados, bem como, os estilos arquiteturais que favorecem tais atributos e as representações das visões arquiteturais e seus sub-produtos.

1.3 Definições, Acrônimos e Abreviações

AAS: Artefato de Arquitetura de Software.

RAS: Requisito de Arquitetura de Software.

Id.: Identificador.

Software: Conjunto de documentações, guias, metodologias, processos, códigos e ferramentas para a solução de um problema.

Sistema: Conjunto de pessoas, softwares, hardwares e outros sistemas para a solução de um problema.

Stakeholder: Indivíduo, grupo ou organização que possua interesse no Sistema.

Visão Arquitetural: Produto resultante da interpretação de um Stakeholder do sistema.

Ponto de Vista Arquitetural: Produto resultante da execução de uma Visão Arquitetural.

Arquitetura de Software: Forma como os componentes são agrupados com o objetivo de construir um software ou sistema.

Trade-Off: Cada arquitetura de software possui seus atributos de qualidade que são favorecidos e desfavorecidos, o trade-off consiste em ter a consciência dessas características para escolher uma arquitetura que favorece os atributos de qualidade priorizados.

Atributos de qualidade: São atributos que impactam diretamente na concepção de um software, são definidos conforme a ISO-IEEE 9126.

Cerne: Funcionalidade-chave / núcleo do software.

Sistema Operacional: Software responsável por gerenciar e abstrair a interação entre o usuário e o hardware ou aplicações externas e o hardware.

UML: Sigla para Linguagem de Modelagem Unificada.

HTML: Sigla para Linguagem de Marcação de Hipertexto. 1

HTTP: Sigla para Protocolo de Transferência de Hipertexto.

Json: Sigla para Notação de Objetos Javascript.

REST: Sigla para Representational State Transfer.

DAO: Sigla para Data Access Object.

Nó-físico: Termo para representar um componente físico de modo geral, como

um navegador ou um banco de dados, por exemplo.

1.4 Referências

Id. Nome do artefato

AAS\_1 **Transferência do Cuidado de Pacientes 3.0** - Documento de Requisitos

AAS\_2 ISO-IEEE 9126

AAS\_3 ISO-IEEE 42010

AAS\_4 Slides Ministrados em Sala

AAS\_5 4+1 View

1.5 Visão Geral

Os próximos tópicos descrevem quais serão os requisitos e restrições utilizados para definir a arquitetura a ser implementada, bem como, quais atributos de qualidades serão priorizados e o porquê da escolha. Quais os padrões arquiteturais serão utilizados conforme os atributos de qualidade selecionados e como funcionará o trade-off entre esses padrões arquiteturais, bem como o porquê da escolha dos padrões arquiteturais. Quais e como as visões arquiteturais serão detalhadas e quais os pontos de vista da arquitetura serão utilizados para descrever as visões.

2. Contexto da Arquitetura

2.1 Funcionalidades e Restrições Arquiteturais

Grande parte dos RAS citados na tabela acima são referentes á requisitos

não-funcionais (ou restrições) definidas pelo documento AAS\_1. Os RAS serão os

responsáveis por guiar as decisões sobre quais estilos arquiteturais serão

adequados para favorecer os atributos de qualidade priorizados.

O RAS\_2 deixa explícito que o contexto da aplicação é Web, enquanto que

os RAS\_4 e RAS\_5 deixam claro a existência de componentes, um cliente e um

servidor. Eles direcionam para a utilização do estilo arquitetural Cliente-Servidor

e fica claro, também, a necessidade da utilização do estilo arquitetural de

Componentes, como forma de viabilizar a separação das responsabilidades do

sistema e do estilo arquitetural REST para definir os métodos de comunicação

entre os componentes.

Os RAS\_3, RAS\_6 e RAS\_7 revelam que às funcionalidades existentes no

software deverão funcionar de forma independente, ou seja, a responsabilidade

para executar uma funcionalidade é apenas do componente que a implementa

Para que seja possível a execução destes RAS é necessário implantar uma

arquitetura modularizada, como preconizado pelo estilo em Camadas. Essa é

uma excelente maneira de se separar as responsabilidades do software, bem

como uma excelente maneira de interação entre o software e o ecossistema nele

inserido.

O RAS\_1 mostra que devido a necessidade de manipular dados

disponibilizados pelo CERCOMP e ao mesmo tempo para estar em consonância

com o RAS\_8 e RAS\_6, a base de dados deverá ser armazenada em ambiente

próprio e será um dos componentes da aplicação. Os outros componentes são

Cliente e Servidor, em consonância com os RAS\_4 e RAS\_5.

Dessa forma, os RAS\_4, RAS\_5 e RAS\_8 mostram que o ambiente para

execução dos componentes é de responsabilidade dos usuários. O CERCOMP no

caso do componente Servidor e do componente de Dados e o usuário-final no

caso do componente Cliente.

2.2 Atributos de Qualidade Prioritários

Atributos de qualidade prioritários, como **segurança** e **usabilidade**, são fundamentais para uma arquitetura de software bem sucedida. De acordo com o documento de estilo de arquitetura de software da IEEE, "os principais atributos de qualidade devem ser considerados ao projetar a arquitetura de software, e podem incluir segurança, usabilidade, escalabilidade, desempenho, fiabilidade, extensibilidade, flexibilidade e manutenibilidade" (IEEE, 2005, p. 4).

Em particular, a segurança é um dos principais atributos de qualidade de um sistema de software, pois garante que os dados e recursos sejam acessados somente por usuários autorizados. Além disso, as arquiteturas de software seguras devem ser projetadas para resistir a ataques maliciosos, garantindo que os usuários não possam acessar dados confidenciais ou modificar recursos não autorizados (Phillips e Rabhi, 2016).

Por outro lado, a usabilidade também é um aspecto importante na arquitetura de software. Esta característica se refere à facilidade de uso do sistema por parte dos usuários. As arquiteturas de software usáveis devem incluir interfaces intuitivas e fáceis de usar, além de permitir aos usuários acessar os recursos de maneira rápida e eficiente (Phillips e Rabhi, 2016).

Ao escolher segurança e usabilidade como atributos de qualidade prioritários na arquitetura de microsserviços, existem alguns trade-offs a serem considerados. Em particular, a segurança é fundamental para garantir que os dados e recursos sejam acessados somente por usuários autorizados, mas isso pode resultar em um desempenho reduzido se a segurança for exagerada. Além disso, a segurança pode aumentar o custo de desenvolvimento, pois implica um esforço maior na implementação de medidas de segurança (Phillips e Rabhi, 2016).

Por outro lado, a usabilidade é um aspecto importante na arquitetura de microsserviços, pois garante que os usuários possam acessar os recursos de maneira rápida e eficiente. No entanto, em alguns casos, a usabilidade pode ser sacrificada em favor de outros atributos de qualidade, como segurança, escalabilidade e desempenho (Phillips e Rabhi, 2016).

Ao escolher segurança e usabilidade como atributos de qualidade prioritários na arquitetura de microsserviços, existem alguns trade-offs a serem considerados. Por um lado, a segurança é importante para garantir que os dados e recursos sejam acessados somente por usuários autorizados, mas isso pode resultar em um desempenho reduzido se a segurança for exagerada. Por outro lado, a usabilidade é importante para garantir que os usuários possam acessar os recursos de maneira rápida e eficiente, mas em alguns casos pode ser sacrificada em favor de outros atributos de qualidade.

Assim, segurança e usabilidade são atributos de qualidade prioritários essenciais para as arquiteturas de software bem-sucedidas, pois garantem que os dados e recursos sejam acessados somente por usuários autorizados e que os usuários possam acessar os recursos de maneira rápida e eficiente.

3. Representação da Arquitetura

Descrição sobre a arquitetura CQRS

O CQRS (Segregação de Responsabilidade de Comando e Consulta), é um padrão de arquitetura que separa as operações de leitura e atualização de um banco de dados. A implementação do CQRS em seu aplicativo pode maximizar o desempenho, a escalabilidade e a **segurança**. A flexibilidade criada pela migração para CQRS permite ao sistema evoluir melhor ao longo do tempo e impede que os comandos de atualização causem conflitos de mesclagem no nível de domínio.

O uso da arquitetura CQRS (Segregação de Responsabilidade de Comando e Consulta) tem se tornado cada vez mais comum na área de desenvolvimento de software. Esta abordagem é baseada na ideia de que os comandos e as consultas devem ser tratados de forma separada, permitindo que as responsabilidades de cada um sejam claramente definidas. Como afirmou Martin Fowler: 'CQRS é uma arquitetura que separa os comandos que mudam o estado de um sistema de suas consultas para os dados desse sistema'. Esta separação permite que o código seja mais simples, o que resulta em melhor desempenho e escalabilidade. Além disso, como afirmou Greg Young, 'CQRS nos dá a capacidade de nos concentrarmos em cada camada de forma independente, permitindo que cada camada seja otimizada para sua própria responsabilidade'. Portanto, CQRS é uma abordagem poderosa para desenvolver aplicativos modernos e escaláveis.

O CQRS separa as leituras e gravações em modelos separados, usando comandos para atualizar dados e consultas para ler dados.

* Os comandos devem ser baseados em tarefas, em vez de centrados nos dados. ("Book hotel room", não "set ReservationStatus to Reserved").
* Os comandos podem ser colocados em uma fila para processamento assíncrono, em vez de serem processados de forma síncrona.
* As consultas nunca modificam o banco de dados. Uma consulta retorna um DTO que não encapsula qualquer conhecimento de domínio.

Segue abaixo uma ilustração representando o padrão arquitetural CQRS

(imagem)

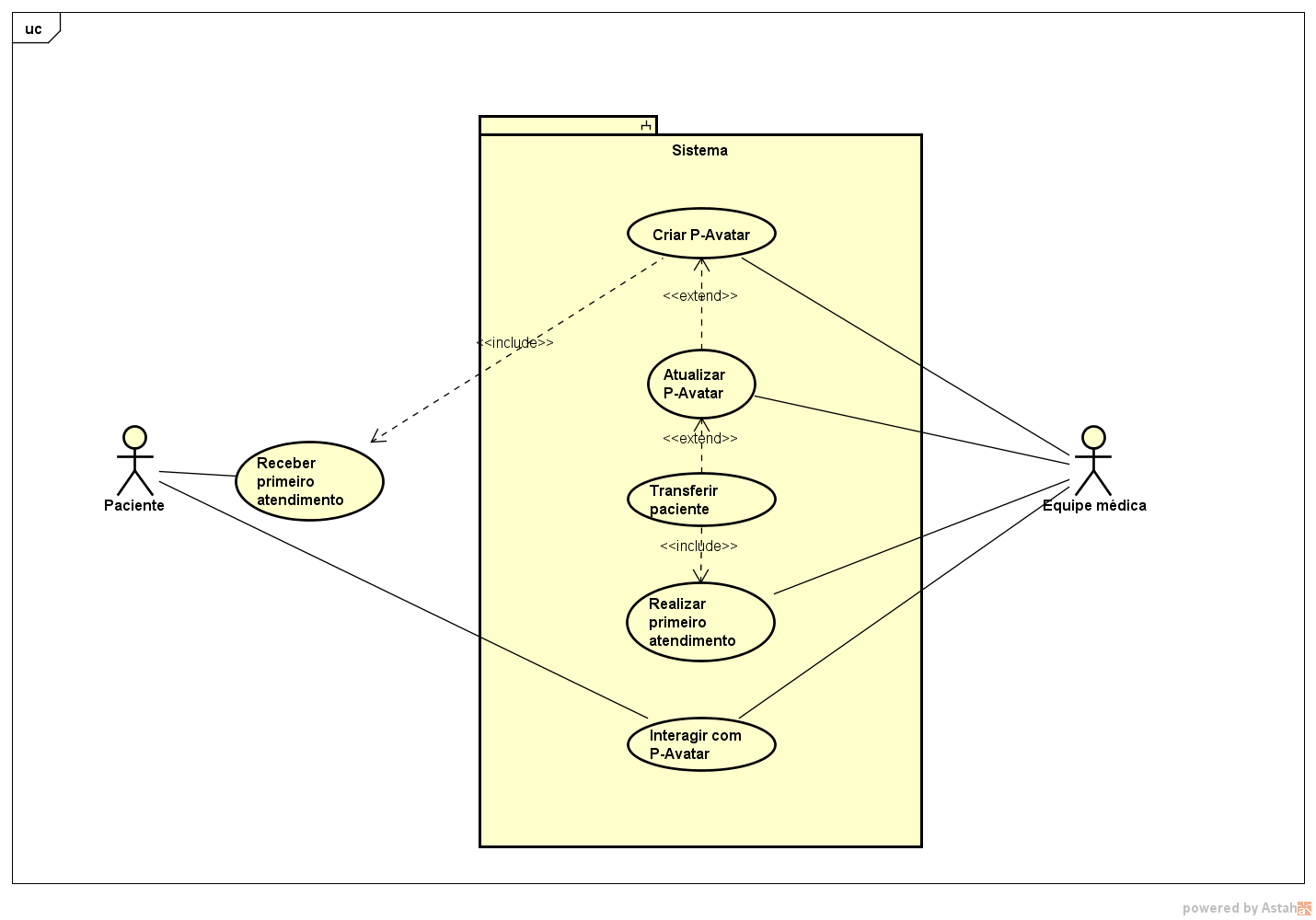
4. Ponto de vista dos Casos de Uso

4.1 Descrição

4.2 Visão de Casos de Uso

Cada requisito funcional definido em AAS\_1 foi considerado um caso de uso e analisado de forma a gerar o diagrama de casos de uso do software a ser desenvolvido.

Imagem 2 - Diagrama de Caso de Uso



fonte: Elaborado pelos autores.

5

O metamodelo do diagrama de componentes está localizado no Anexo A -

Metamodelo de Casos de Uso.

5. Ponto de vista do Projetista

5.1 Visão Geral

O ponto de vista do projetista é direcionado aos projetistas e

desenvolvedores do software e tem como objetivo definir as principais partes que

o compõem, tal como os componentes, além de definir quais as suas

responsabilidades. Foi escolhida por ser uma visão primordial para a

compreensão do software e de todo o seu ecossistema.

O modelo arquitetural proposto para a construção deste software será

composto por 3 (três) componentes essenciais: Um cliente, um servidor e um

componente responsável por gerenciar os dados utilizados no software.

5.2 Visão de Componentes

O componente cliente é responsável por interagir diretamente com o

usuário, ele é representado pelo navegador, é responsável por realizar a

comunicação com o componente servidor e é responsável por prover uma

camada inicial de segurança do software por meio de autenticação.

O componente servidor é responsável pela implementação da lógica

presente no software, possuindo um conjunto de 5 (cinco) camadas internas para

a segregação das responsabilidades. Cada camada é independente, sendo

responsável por garantir a segurança e execução de suas próprias

6

funcionalidades, sendo acessadas apenas pela camada exterior e possuindo

acesso apenas à camada interior, exceto camadas com acesso direto a

dispositivos de entrada / saída.

O componente fonte de dados é uma representação dos diversos sistemas

externos ao software responsáveis por disponibilizar os dados necessários para o

funcionamento do sistema conforme definido pelo RAS\_1.

A interação entre os componentes citados neste tópico pode ser

visualizada por meio do Diagrama de componentes UML abaixo:

O metamodelo do diagrama de componentes está localizado no Anexo B -

Metamodelo de Componentes.

5.3 Detalhamento das Camadas

Conforme citado no tópico anterior (5.2) : O componente servidor é

dividido internamente por 5 (cinco) camadas, cada camada funciona de forma

independente e possui suas próprias responsabilidades.

A camada View é responsável por prover uma interface para acesso ao

servidor, seja pela exibição de uma página html ou pela utilização de outra

tecnologia. A camada View se comunica apenas com a camada interior Router.

A camada Router é responsável por tratar requisições HTTP, seja

redirecionando o usuário para outra página, seja enviando um objeto Json ao

requerente. A camada Router se comunica apenas com as camadas View e

Presenter.

A camada Presenter é responsável pelo gerenciamento das

funcionalidades do software, tais como: gerar gráficos estatísticos, gerenciar

tempo dos usuários etc. Além de ser responsável pela comunicação com uma

interface para acesso ao banco de dados . A camada Presenter se comunica

apenas com as camadas Router e Extractor.

A camada Extractor é responsável pela interação entre o servidor as

múltiplas fontes de dados utilizados. Essa camada utiliza dos padrões

arquiteturais Factory e DAO para abstrair a forma de obtenção dos dados. A

camada Extractor se comunica apenas com as camadas Entity e Presenter.

E por fim, camada Entity é responsável por gerenciar as entidades

atômicas do software, tais como: usuário, localização, formulários etc. A camada

Entity se comunica apenas com a camada Extractor.

Devido a independência entre as camadas, elas podem ser facilmente

substituídas e validadas caso sejam necessárias, satisfazendo os atributos de

qualidade internas propostas pelo artefato AAS\_2 para a manutenibilidade do

sistema: Analisabilidade, Modificabilidade, Estabilidade e Testabilidade.

A comunicação entre o componente servidor e o ecossistema, e o servidor

e suas camadas internas, se dá por meio de interfaces de acesso, seja

provendo-as ou consumindo-as, como o ocorrido nas camadas View e Extractor.

Devido a essas interfaces, os atributos de qualidade interna propostas pelo

artefato AAS\_2 para a portabilidade do sistema é satisfeito: Adaptabilidade,

Capacidade para ser instalado, Coexistência e Capacidade para substituir.

6. Ponto de vista do Desenvolvedor

6.1 Visão Geral

O ponto de vista do desenvolvedor é direcionado aos projetistas e

desenvolvedores do software e tem como objetivo definir as principais partes

responsáveis por definir as funcionalidades e restrições do software, tal como as

classes.

6.2 Visão lógica

A visão lógica é responsável por definir como a estrutura dos componentes

do software será realizada e foi escolhida para auxiliar na construção da

arquitetura proposta.

6.2.1 Detalhamento das classes

Conforme definido anteriormente (5.3) o componente servidor é dividido

internamente em camadas, cada qual com responsabilidades distintas. O modelo

construído visa representar o componente servidor e sua interação com o

ecossistema proposto.

Para a camada Entity, possuímos as seguintes classes: User, responsável

por representar um usuário do sistema, bem como, suas credenciais e as

interfaces para acesso InterfaceAuthentication, responsável por abstrair a

autenticação no sistema, InterfaceRouter, responsável por abstrair as rotas do

software, e InterfaceFactory, responsável por abstrair a instanciação dos objetos

atômicos e do acesso às fontes de dados.

Para a camada Provider, possuímos as exceções

InvalidParameterException e MethodNotImplementedException, responsáveis

por definir o encapsulamento dos possíveis erros e a classe ExceptionCollector

responsável por gerenciar as exceções capturadas.

Para a camada Extractor, possuímos a implantação do padrão de projeto

de software Factory por meio da classe UserFactory que implementa a interface

InterfaceFactory e é responsável por instanciar as classes atômicas do software,

com exceção das classes com “Exception” no nome.

Para a camada Routes, possuímos a utilização da interface InterfaceRouter

que é implementada pelas classes AuthRouter, e IndexRouter, que é responsável

por gerenciar as rotas do software.

A interação entre as classes citadas neste tópico pode ser visualizada por

meio do Diagrama de classes UML abaixo:

9

O metamodelo do diagrama de classes está localizado no Anexo C - Metamodelo

de Classes.

6.3 Visão de segurança

Conforme definido anteriormente (2.2), a segurança é um dos atributos de

qualidade a serem priorizados durante o desenvolvimento da arquitetura do

software. A visão de segurança é responsável por definir como a segurança dos

usuários e do software será realizada à nível estrutural e foi escolhida devido à

natureza do software e sua intrínseca relação com o gerenciamento e análise de

dados.

Uma das características de se construir visando a independência entre os

componentes é a implementação de níveis de segurança para cada componente.

O componente servidor faz o uso dessa característica para as camadas internas,

em que a comunicação entre as camadas se dá por meio da implementação de

interfaces, conforme definido na visão anterior (6.2).

10

Entretanto cada camada interna deverá realizar o tratamento de suas

próprias exceções, conforme o tópico a seguir.

6.3.1 Detalhamento da segurança

As classes App, AuthRoute e UserFactory são responsáveis pela captura

das exceções de suas camadas inferiores. A classe App é responsável durante sua

inicialização de middleware por verificar se as rotas de acesso ao sistema foram

geradas corretamente, enquanto que a classe AuthRoute é responsável por

verificar se o processo de autenticação está funcionando corretamente, e por fim,

a classe UserFactory é responsável por instanciar os usuários do sistema ou

realizar a comunicação com o(s) banco(s) de dados corretamente.

No caso de uma falha em qualquer uma das etapas citadas acima, uma

exceção é gerada ela é coletada pela classe ExceptionCollector e tratada em

seguida.

A interação entre a segurança pode ser visualizado no diagrama UML

abaixo:

11

O metamodelo do diagrama de classes está localizado no Anexo D -

Metamodelo de Fluxo de exceções.

7. Ponto de vista do Implantador

7.1 Visão Geral

A visão de implantação é direcionada para a equipe de implantação e é

responsável por definir as ferramentas e ambiente necessário para o bom

funcionamento do software. Ela foi escolhida por se tratar de um software com

múltiplos componentes independentes e a necessidade de coexistir com um

ecossistema potencialmente mutável.

7.2 Visão física

Conforme o tópico 5.1 há 3 (três) componentes essenciais que devem ser

levados em consideração durante uma visão arquitetural: Um componente

cliente, um componente servidor e um componente para acesso aos dados.

Os componentes citados são nós-físicos do software, cada qual com sua

própria configuração e ferramentas necessárias para o funcionamento e foi

escolhida para auxiliar na visualização dos componentes externos ao sistema tais

como: hardware, sistema operacional e softwares externos.

7.2.1 Detalhamento dos nós-físicos

O nó-físico do cliente é o computador do próprio usuário e para o seu

funcionamento é necessário a instalação dos navegadores Google Chrome versão

48 ou Firefox versão 44 por serem os softwares responsáveis por acessar o

sistema a ser desenvolvido.

O nó-físico do servidor do software a ser desenvolvido é uma máquina,

física ou virtual, com sistema operacional Open Suse versão 12.3 é necessário a

instalação do software Node versão 8.11.3 com a dependência expressJs versão

4.15.2 para a construção, funcionamento e publicação do servidor em ambiente

operacional. É necessário também a utilização do software docker versão 18.03.1 -

ce para a abstração do sistema operacional, facilitando a implantação do software

na máquina disponibilizada, além das dependências: mocha, chai e chai-http nas

respectivas versões 4.0.1, 4.1.2 e 3.0.0 para a realização e execução de testes

unitários automatizados; e do software externo wrk para a execução de testes de

carga.

O nó-físico de dados é uma abstração das aplicações externas ao software

que são responsáveis por disponibilizar os dados utilizados para o funcionamento

12

do software. O tipo do banco de dados utilizado é o PostgreSQL versão 9.5 ou o

MariaDB versão 10.1.

A comunicação entre o nó-físico cliente e o nó-físico servidor se dá por

meio do protocolo http, enquanto que a comunicação entre o nó-físico servidor e

o nó-físico de dados por meio do mapeador objeto relacional sequelize que

acessa os dados por requisições http.

A interação entre os nós físicos citados neste tópico pode ser visualizada

por meio do Diagrama de implantação UML abaixo:

O metamodelo do diagram

Referências

IEEE. (2005). IEEE Standard for Software Architecture Description. IEEE Standard 1471-2000. Recuperado de https://ieeexplore.ieee.org/document/1145796

Phillips, T. E. Rabhi, F. (2016). Projeto de software: Teoria e prática. 5ª ed. Rio de Janeiro: Elsevier.

Fowler, M. (2011). CQRS. Consultado em 9 de dezembro de 2022, de https://martinfowler.com/bliki/CQRS.html

Young, G. (2008). CQRS, Task Based UIs, Event Sourcing agh! Consultado em 9 de dezembro de 2022, de https://codebetter.com/gregyoung/2008/04/09/cqrs-task-based-uis-event-sourcing-agh/