Sempre UFG

Documento de Arquitetura de Software

1. Introdução

1.1 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

SUFG: Sempre UFG.
CERCOMP: Centro de Recursos Computacionais UFG.
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*.

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	SUFG- 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

Id.	Tipo	Id. do Documento de Requisitos SUFG
RAS_1	Requisitos de Dados	*Todos*
RAS_2	Requisitos Não-Funcionais	RNF-AplicWeb
RAS_3	Requisitos Não-Funcionais	RNF-IntgrCercom

RAS_4	Requisitos Não-Funcionais	RNF-AmbOpServ
RAS_5	Requisitos Não-Funcionais	RNF-AmbOpCli
RAS_6	Requisitos Não-Funcionais	RNF-FuncIndep
RAS_7	Requisitos Não-Funcionais	RNF-CapacDados
RAS_8	Requisitos Não-Funcionais	RNF-SegInfo

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 Qualidades Prioritários

Conforme definido pelo tópico anterior (**2.1**), o software a ser desenvolvido realizará interações com os dados obtidos pelo *CERCOMP*. Devido a essa característica é necessário implantar um modelo arquitetural que possua a **Segurança** como um de seus atributos de qualidade.

Tal atributo pode ser favorecido pelos estilos arquiteturais camadas e componentes que devido a sua modularização permite que diversos níveis de segurança sejam implementados. Entretanto há um crescimento da complexidade quando comparamos aos demais estilos o que resulta resulta em certa redução na eficiência, por *Trade-off*.

Os estilos arquiteturais, camadas, componentes e cliente-servidor, também favorecem o atributo de qualidade **Manutenibilidade**, pois permitem que os componentes possam ser facilmente substituídos caso haja necessidade, devido a divisão das responsabilidades.

Segundo o tópico de portabilidade do *AAS_1*:

"*O CERCOMP será responsável pela adaptação do ambiente operacional físico e de hardware para atender às necessidades do software, incluindo a disponibilidade de máquina (física ou virtual) com capacidade de hardware para processamento e comunicação de dados e a disponibilidade de recursos e canais de comunicação com a Internet. A equipe de desenvolvimento do software será responsável por configurar a máquina fornecida pelo CERCOMP para atender as necessidades da plataforma lógica do software.*"

"O componente cliente deve ser utilizado em condições ambientais físicas típicas de escritórios, empregando computadores convencionais (desktop) com tela padrão de 17 polegadas e resolução de 1920 x 1080 pixels. Cabe ao próprio usuário do software prover o ambiente físico para execução do componente cliente do software."

O Software a ser desenvolvido deverá coexistir com outros sistemas. Os estilos arquiteturais cliente-servidor e componentes, em conjunto com ferramentas de implantação (vide tópico **5** para o detalhamento das tecnologias a serem utilizadas), favorecem a coexistência do software e por conseguinte o atributo de qualidade **Portabilidade**. Não obstante, o impacto do *Trade-Off* é sentido no que tange à eficiência, devido a latência da rede e o aumento da complexidade.

E assim, definimos os atributos de qualidade a serem priorizados durante o desenvolvimento da arquitetura a ser implementada: Segurança, Manutenibilidade e Portabilidade respectivamente.

3. Representação da Arquitetura

Conforme definido pelos tópicos anteriores (**2.1** e **2.2**), a arquitetura do software a ser desenvolvido será uma arquitetura híbrida e independente que

une as principais características dos estilos arquiteturais: *Camadas*, *Cliente-Servidor, REST* e *Componentes, e* prioriza os Atributos de qualidade: Segurança, Manutenibilidade e Portabilidade.

Para representar as decisões arquiteturais definidas ao findar da análise, serão utilizados os pontos de vista definidos nos documentos *AAS_4* e *AAS_5*, que são:

Ponto de Vista	Visão	Diagrama(s)
Projetista	**Desenvolvimento**	Componentes
Desenvolvedor	**Lógica**	Classes
	Segurança	Pontos-fracos
Implantador	**Física**	Implantação
-	**Casos de Uso**	Casos de uso

Os tópicos a seguir detalham esses pontos de vista arquiteturais, bem como as visões e os metamodelos utilizados para representá-los.

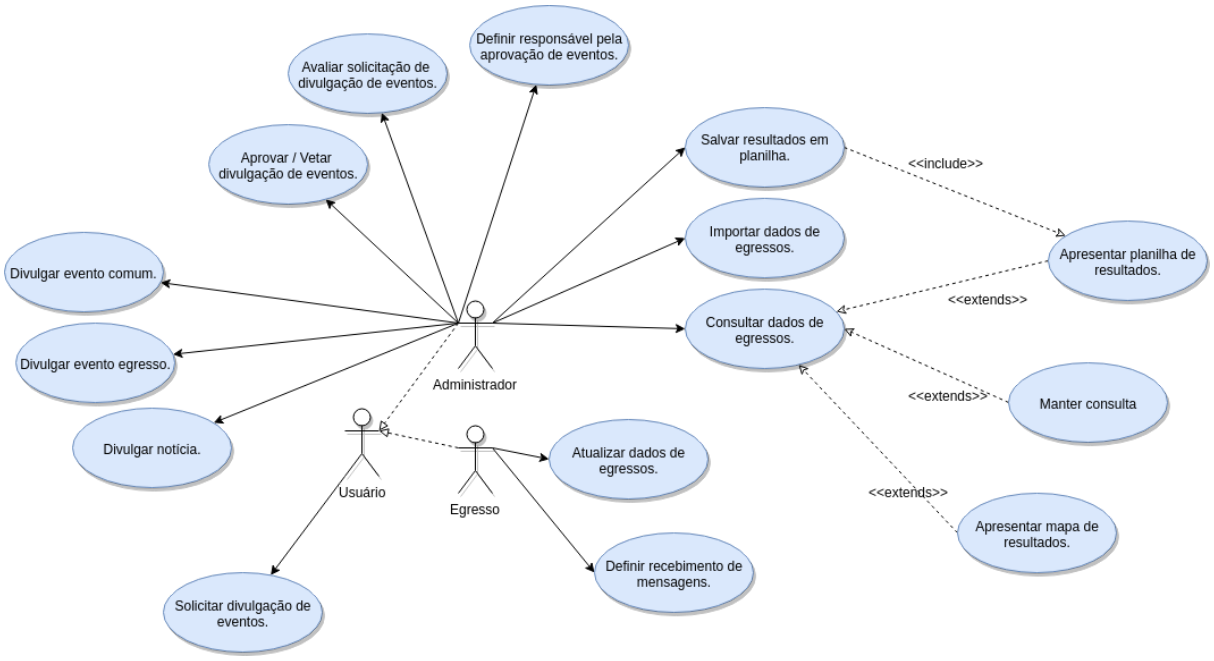
4. Ponto de vista dos Casos de Uso

4.1 Descrição

Para fornecer uma base para o planejamento da arquitetura e de todos os outros artefatos que serão gerados durante o ciclo de vida do software, é gerada, na análise de requisitos, uma visão chamada **visão de casos de uso**. Só existe uma visão de casos de uso para cada sistema. Ela ilustra os casos de uso e cenários que englobam o comportamento, as classes e riscos técnicos significativos do ponto de vista da arquitetura. A visão de casos de uso é refinada e considerada inicialmente em cada iteração do ciclo de vida do software.

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.



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

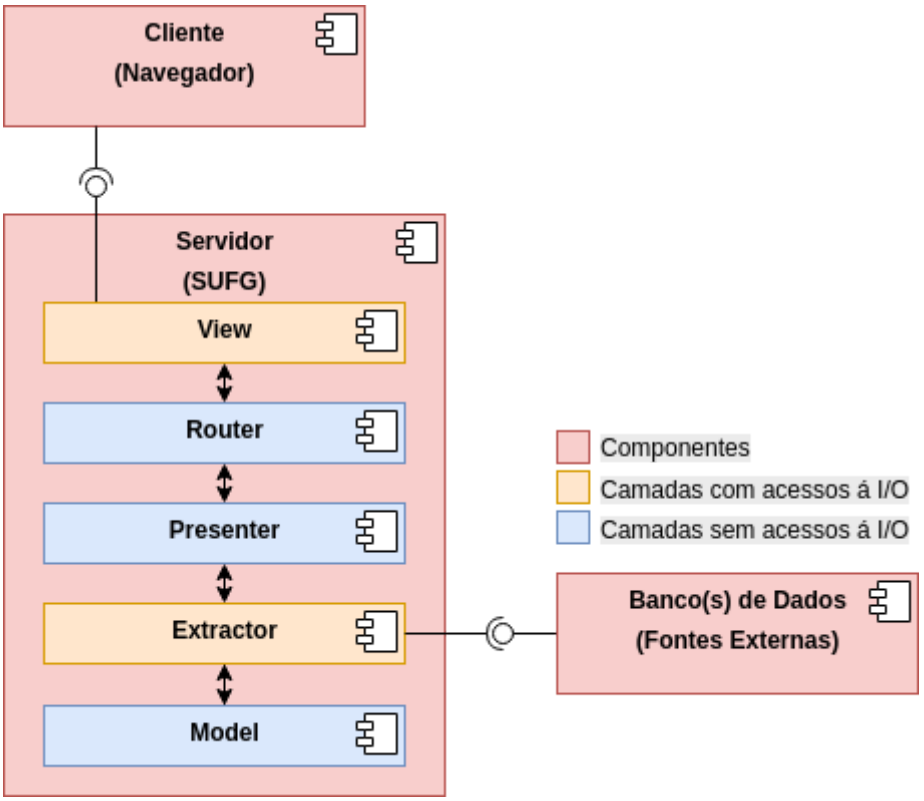
O componente *cliente* é responsável por interagir diretamente com o usuário, ele é representado pelo navegador, é responsável por realizar a comunicação com o componente servidor e é responsável por prover uma camada inicial de segurança do software por meio de autenticação.

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

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.

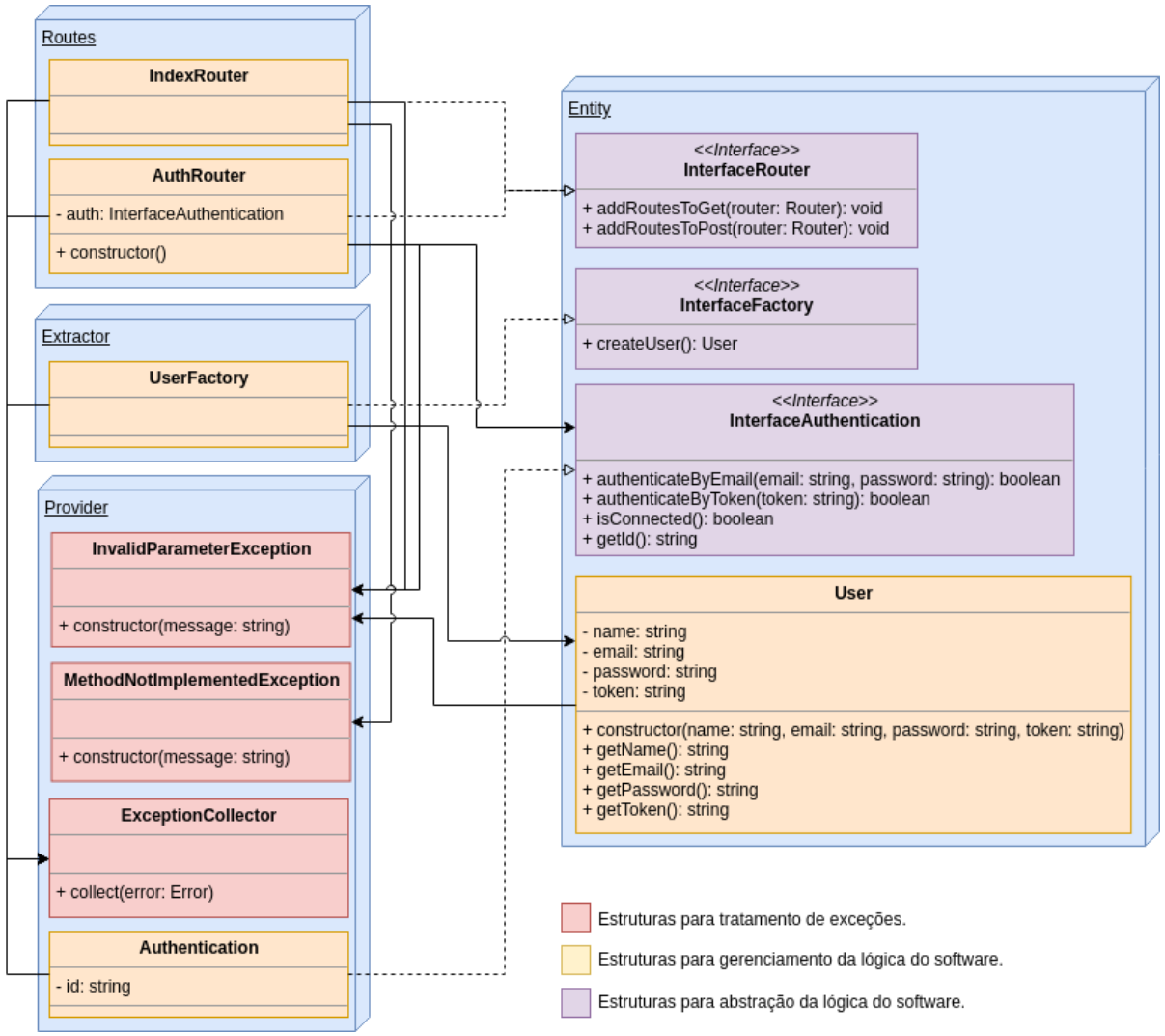
Para a camada Entity, possuímos as seguintes classes: *User*, responsável por representar um usuário do sistema, bem como, suas credenciais e as interfaces para acesso *InterfaceAuthentication, responsável por abstrair a autenticação no sistema, InterfaceRouter, responsável por abstrair as rotas do software, 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:



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**).

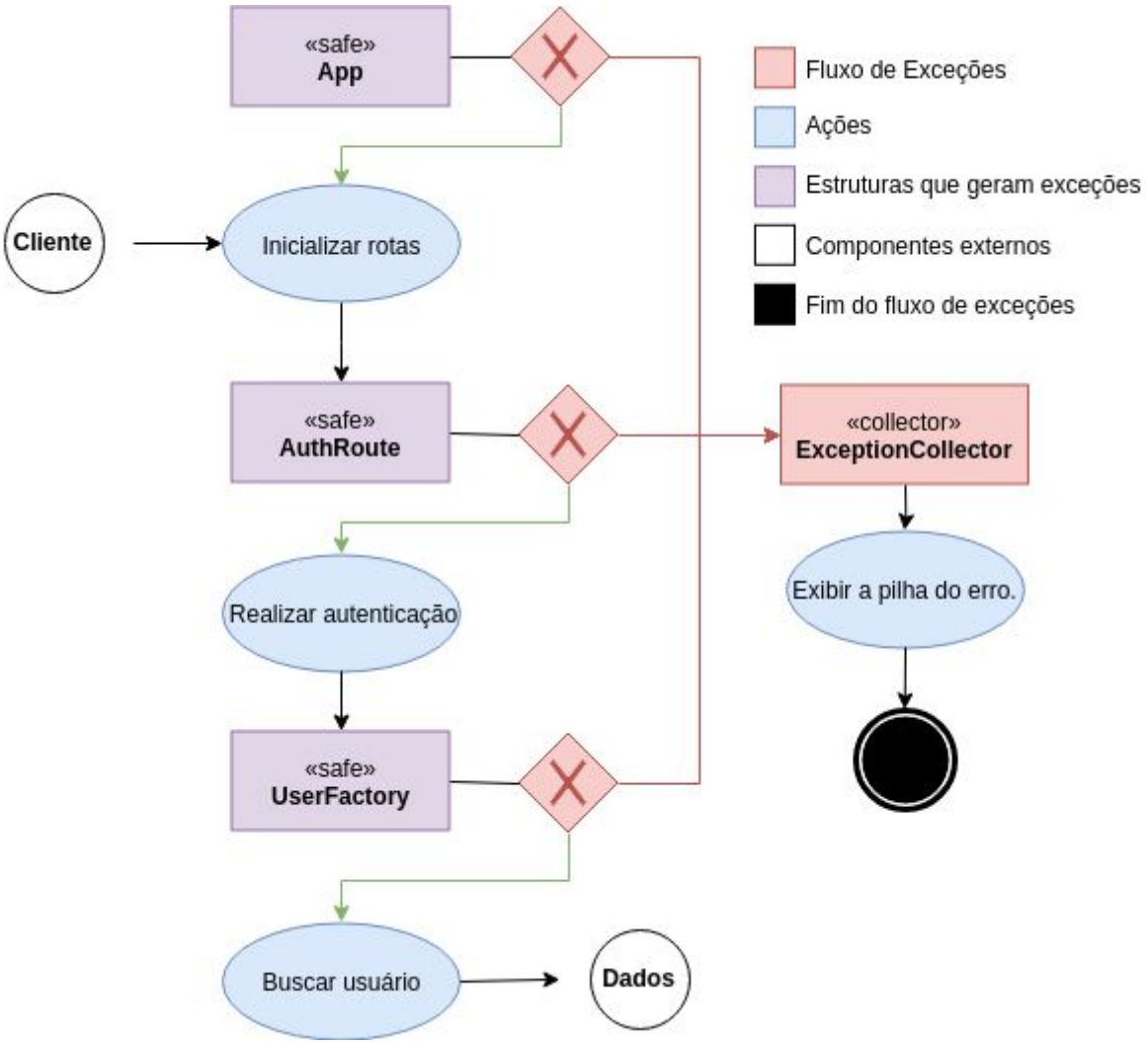
Entretanto cada camada interna deverá realizar o tratamento de suas próprias exceções, conforme o tópico a seguir.

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:



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.

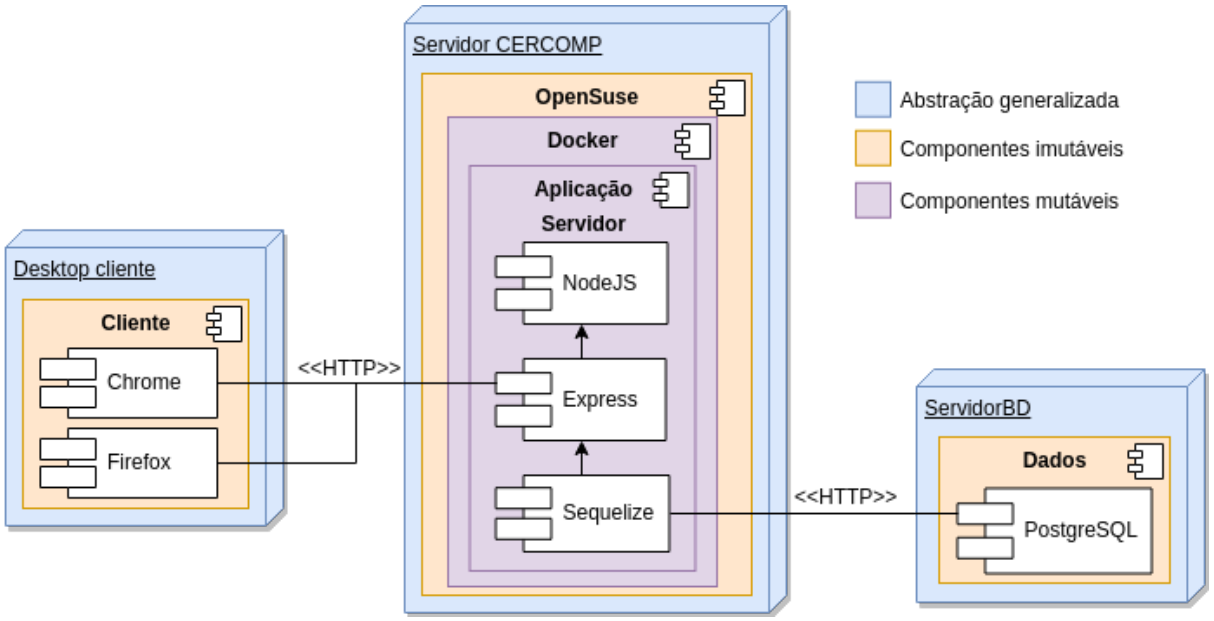
O nó-físico do *servidor* do software a ser desenvolvido é uma máquina, física ou virtual, com sistema operacional Open Suse versão 12.3 é necessário a instalação do software Node versão 8.11.3 com a dependência expressJs versão 4.15.2 para a construção, funcionamento e publicação do servidor em ambiente 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

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 diagrama de implantação está localizado no **Anexo E - Metamodelo de Implantação**.

Anexo A - Metamodelo de Casos de Uso.

A.1 Descrição:

O diagrama de casos de uso é um diagrama utilizado para definir os usuários-finais do sistema e quais funcionalidades poderão ser realizadas por cada usuário-final.

A.2 Estruturas:

O diagrama de casos de uso é composto por 4 (quatro) estruturas básicas:

Atores: São os usuários-finais do sistema. São responsáveis por realizar casos de uso do sistema e podem herdar responsabilidades de outros atores. São representados pelo ícone de um *StickMan*.

Casos de uso: São as funcionalidades realizadas pelo sistema. São análogos aos requisitos funcionais de um software. São representados por um *Oval*.

Cenários: É a sequência de eventos realizados por um caso de uso, pode representar um caso de uso externo de uso obrigatório com o estereótipo <<include>> ou pode representar um caso de uso externo de uso opcional com o estereótipo <<extend>>.

Relacionamento: É a estrutura responsável pela comunicação entre os casos de uso com atores e entre os casos de uso com casos de uso, por meio dos cenários. É representado por uma linha tracejada com ponta em forma de seta branca.

Anexo B - Metamodelo de Componentes.

B.1 Descrição:

O diagrama de componentes é um diagrama utilizado para definir as estruturas a nível macro do sistema e como suas estruturas internas são definidas assim como suas formas de comunicação.

B.2 Estruturas:

O diagrama de componentes é composto por 2 (duas) estruturas básicas:

Componente: Componentes são as estruturas centrais deste metamodelo, eles são responsáveis por representar todas as estruturas importantes do ponto de vista a ser representado, tais como arquivos, bancos de dados, abstrações de camadas etc.

Relacionamento: É a estrutura responsável pela comunicação entre os componentes. É representado de duas formas: Por uma linha contínua com ponta em forma de seta simples, responsável pela comunicação direta entre os componentes. Por uma notação de interface nomeada *lollipop*.

Anexo C - Metamodelo de Classes.

C.1 Descrição:

O diagrama de classes é um diagrama utilizado para definir a implementação a nível micro do sistema, definindo quais são as estruturas principais do sistema assim como suas formas de comunicação.

C.2 Estruturas:

O diagrama de classes é composto por 3 (três) estruturas básicas:

Classe: São as estruturas centrais deste metamodelo, eles são responsáveis por representar os elementos centrais que compõem a implementação do sistema a ser desenvolvido.

Interface: Baseado no paradigma orientado a objetos, uma interface é uma abstração responsável por gerenciar responsabilidades em comum.

Relacionamento: É a estrutura responsável pela comunicação entre as classes com classes e entre as classes com as interfaces. É representado de duas formas: Por uma linha tracejada com ponta em forma de seta branca, responsável por representar uma comunicação entre interfaces e classes. Por uma linha contínua com ponta em forma de seta simples, responsável por representar uma comunicação entre classes com classes.

Anexo D - Metamodelo de Fluxo de exceções.

D.1 Descrição:

O diagrama de fluxo de exceções é um diagrama utilizado para definir a captura e tratamento das exceções geradas pelo sistema, definindo quais são as estruturas principais assim como suas formas de comunicação.

D.2 Estruturas:

O diagrama de fluxo de exceções é composto por 5 (cinco) estruturas básicas:

Ação: São as ações realizadas durante o fluxo de eventos do software. É representado por um oval.

Módulos de segurança: São as estruturas do software capazes de gerar exceções, são representados pelo estereótipo *<<safe>>* e possuem uma estrutura condicional.

Módulos de captura: São as estruturas responsáveis por tratar exceções do software, são representados pelo estereótipo *<<collector>>*.

Condicionais: São estruturas responsáveis por representar uma captura de exceção, é representada na forma de um losango com um **X** e é dividida por duas setas, uma para expressar o fluxo normal e outra para expressar o fluxo com exceção.

Relacionamento: É a estrutura responsável pela comunicação entre os módulos e as ações. É representado de duas formas: Por uma linha verde contínua com ponta em forma de seta simples, responsável por representar um fluxo normal. Por uma linha vermelha contínua com ponta em forma de seta simples, responsável por representar um fluxo de exceção.

Anexo E - Metamodelo de implantação.

E.1 Descrição:

O diagrama de implantação é um diagrama utilizado para definir quais os softwares e hardwares auxiliares utilizados pelo sistema, descrevendo a interação entre eles.

E.2 Estruturas:

O diagrama de implantação é composto por 2 (duas) estruturas básicas:

Componente: São os softwares e hardwares auxiliares utilizados pelo sistema a ser representado.

Relacionamento: É a estrutura responsável pela comunicação entre os componentes. É representado por uma linha contínua sem ponta, responsável por representar a comunicação entre os componentes, pode ser acompanhado pela nomenclatura do tipo da comunicação.