PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS

ESCOLA POLITÉCNICA E DE ARTES

GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Arquitetura de Software

LIANDRA DE OLIVEIRA LOPES

GOIÂNIA
2023

LIANDRA DE OLIVEIRA LOPES

Arquitetura de Software

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, como parte dos requisitos para a obtenção do título de Bacharel em Ciência da Computação.

Orientador(a): Prof. Me. Eugênio Júlio Messala Cândido Carvalho

GOIÂNIA
2023

AGRADECIMENTOS

Agradeço primeiramente a Deus pela vida e pelas oportunidades que me foram concedidas. Aos meus pais, em especial a minha mãe Patrícia e minha avó que ficaram extremamente felizes com a minha entrada na universidade, mas infelizmente não poderão estar para me ver terminando. A minha tia/madrinha Carla que sempre me ensinou a correr atrás dos meus sonhos e ser independente.

Agradeço aos meus amigos e colegas de faculdade por todo apoio e suporte durante esses anos. A minha irmã de coração e alma Ana Beatriz, e meu melhor amigo Caio por aguentarem todos os meus surtos até aqui. Aos meus professores e ao meu orientador Eugênio em especial, por sempre me guiar e tornar essa experiência um pouco mais legal e tranquila. Agradeço também ao Lucas Canedo, por ser meu porto seguro e sempre estar do meu lado.

Agradeço também a todos que me apoiaram em todo esse processo de aprendizagem e que me ajudaram a chegar até aqui. Meu muito obrigada.

RESUMO

Neste trabalho, é apresentada uma pesquisa sobre alguns estilos arquiteturais, seus benefícios e como impactam na construção de software, tendo como objetivo principal apresentar fundamentos que apontem o uso dos modelos de arquitetura e facilitam o desenvolvimento de sistemas. A arquitetura de software é o processo de definição da estrutura de um sistema de software. Ela define os componentes do sistema, como eles se comunicam entre si e como eles interagem com o ambiente externo. A qualidade do software é determinada por uma série de fatores, como a confiabilidade, a segurança, a usabilidade e a eficiência. A arquitetura de software pode influenciar a qualidade do software de várias maneiras. Por exemplo, uma arquitetura bem projetada pode facilitar o desenvolvimento de código, tornar o software mais fácil de usar, melhor o desempenho e manutenção do software. Os modelos arquiteturais são representações abstratas da arquitetura de um sistema de software e sua escolha depende de uma série de fatores, como por exemplo, o tamanho e a complexidade do sistema, restrições de orçamento e tempo, ou se precisam ser escaláveis.

Palavras-chaves: Arquitetura de Software. Modelos Arquiteturais. Estilos Arquiteturais.

ABSTRACT

In this work, a research on some architectural styles, their benefits, and how they impact software construction is presented, with the main goal of providing foundations that point towards the use of architectural models and facilitate system development. Software architecture is the process of defining the structure of a software system. It outlines the components of the system, how they communicate with each other, and their interaction with the external environment. Software quality is determined by factors such as reliability, security, usability, and efficiency. Software architecture can influence software quality in various ways. For example, well-designed architecture can facilitate code development, make the software more user-friendly, and improve software performance and maintenance. Architectural models are abstract representations of the architecture of a software system, and their selection depends on various factors such as the size and complexity of the system, budget and time constraints, or the need for scalability.

Keywords: Software Architecture. Architectural Models. Architectural Styles.

LISTA DE FIGURAS

SUMÁRIO

1. INTRODUÇÃO

A arquitetura de software é um elemento-chave da engenharia de software e desempenha um papel fundamental no projeto, organização e construção de sistemas de software complexos. No contexto do contínuo desenvolvimento tecnológico e da crescente procura por sistemas eficientes e flexíveis, a arquitetura de software torna-se um fator decisivo no sucesso dos projetos de desenvolvimento. Este trabalho fornece uma pesquisa dos princípios, conceitos e modelos relacionados à arquitetura de software, enfatizando sua importância na construção de soluções de software eficientes e escaláveis.

Ao longo das últimas décadas, o campo da arquitetura de software passou por significativas transformações, refletindo não apenas as mudanças nas tecnologias disponíveis, mas também a complexidade crescente das demandas colocadas sobre os sistemas de software modernos. Neste contexto, é importante não só compreender os fundamentos da arquitetura de software, mas também explorar os vários modelos arquiteturais que surgiram para resolver desafios específicos em diferentes domínios e cenários de aplicação.

Este trabalho abordará os conceitos iniciais da arquitetura de software, proporcionando uma visão abrangente dos princípios que norteiam a criação de sistemas de software eficazes. Além disso, serão analisados e comparados alguns modelos arquiteturais, tais como a arquitetura monolítica, arquitetura em camadas e microsserviços. Cada modelo será examinado em termos de suas vantagens, desvantagens e casos de uso ideais, proporcionando um entendimento aprofundado das escolhas arquiteturais.

2. OBJETIVO GERAL

O objetivo desse trabalho é apresentar os conceitos iniciais da arquitetura de software e fundamentos que apontem o uso dos modelos de arquitetura e como facilitam o desenvolvimento de sistemas.

2.1. Objetivos Específicos

- Realizar uma pesquisa bibliográfica sobre o conceito de arquitetura de

software e sua importância;

- Realizar uma pesquisa bibliográfica acerca de alguns modelos de arquitetura e suas características;
- Implementar os modelos de pipes e filtros, monolítico, em camadas e microserviços, identificando suas características, vantagens e desvantagens.

2.2. Procedimentos Metodológicos

Segundo os procedimentos técnicos, essa pesquisa é bibliográfica e experimental. Bibliográfica pois se iniciou com uma pesquisa detalhada em artigos, periódicos e livros sobre arquitetura de software e estilos arquiteturais. A pesquisa também abrangeu características desses modelos e suas vantagens e desvantagens. Também é experimental, pois se faz a observação de um objeto de estudo que tem resultados diferentes considerando o modelo arquitetural.

2.3. Estrutura do Trabalho

Este trabalho está divido em 6 partes:

1. **Arquitetura de software:** que irá trazer alguns conceitos básicos sobre a arquitetura de software e sua importância;
2. **Estilos arquiteturais:** que irá trazer alguns modelos de arquiteturas pesquisados e suas características;
3. **Arquitetura Monolítica:** implementação do estilo arquitetural, trazendo suas vantagens e desvantagens;
4. **Arquitetura em Camadas:** implementação do estilo arquitetural, trazendo suas vantagens e desvantagens;
5. **Arquitetura de Microserviços:** implementação do estilo arquitetural, trazendo suas vantagens e desvantagens;
6. **Ferramental Utilizado**;

3. DESENVOLVIMENTO

3.1. Arquitetura de Software

De acordo com Antônio Mendes (MENDES, 2002), a arquitetura de software é o estudo da organização global dos sistemas de software, bem como do relacionamento entre subsistemas e componentes, onde constitui-se num fator de suma importância no projeto de sistemas de software de grande porte.

Conforme Mendes (MENDES, 2002), a arquitetura de software é uma área relativamente nova dentro da engenharia de software. Esse tópico não havia despertado interesse de pesquisadores e profissionais até o final da década de 1980, quando Mary Shaw (SHAW, 1989) apontou a necessidade de considerar o nível organizacional ou arquitetural dos sistemas. Aproximadamente quatro décadas atrás, um software constituía uma pequena parcela dos sistemas computacionais quando comparado ao hardware. Naquela época, os custos de desenvolvimento e manutenção de softwares eram desprezíveis. Hoje, porém, software é responsável por significativa porção dos sistemas computacionais. (MENDES, 2002).

Diferentemente do uso de técnicas que empregam algoritmos, estruturas de dados e as linguagens de programação que as implementam, o crescimento em escala dos sistemas de software demandou notações para conectar módulos, técnicas para gerenciar configurações e gerenciar versões. Entretanto, à medida que os sistemas se tornavam maiores e mais complexos, outros fatores como desempenho e qualidade, passaram a ser considerados também. Dentro desse contexto, a arquitetura de software entrou em cena de modo a lidar com sistemas grandes e complexos. (MENDES, 2002).

Mendes (MENDES, 2002) diz que cada vez mais é evidente que o processo de engenharia de software requer um projeto de arquitetura de software, e explica isso através dos quatros pontos a seguir:

- É importante ser capaz de reconhecer estruturas comuns utilizadas em sistemas já desenvolvidos, possibilitando aos projetistas de software compreender as relações existentes entre sistemas e desenvolver novos sistemas com base em variações de sistemas antigos;
- O entendimento de arquiteturas de software existentes permite aos engenheiros tomarem decisões sobre alternativas de projeto;
- Uma descrição arquitetural do sistema é essencial a fim de analisar e descrever

propriedades de um sistema complexo;

- O conhecimento de notações formais para descrição de paradigmas arquiteturais permite que o engenheiro possa apresentar novos projetos de sistemas para outras pessoas e instituições.

Conforme a arquitetura de software foi se tornando um tema fundamental na construção de um software, a necessidade e o papel do arquiteto de software também se tornaram importante. Um arquiteto de softwares precisa ter conhecimento profundo do domínio onde o sistema a ser desenvolvido será utilizado, das tecnologias relevantes e dos processos de desenvolvimento. Além disso, o arquiteto de softwares deverá ter um foco nas implicações que os objetivos organizacionais terão sobre as opções técnicas, ou seja, uma visão geral do sistema. (MENDES, 2002).

O arquiteto de softwares estabelece padrões técnicos a serem seguidos durante o desenvolvimento do sistema. Tais padrões incluem a arquitetura do software, a qual engloba definições cruciais para a sua segurança, escalabilidade e performance. Esse profissional apoia a tomada de decisões técnicas durante o desenvolvimento do sistema e trabalha em colaboração com os programadores (MARQUES, 2021).

Parafraseando Martin Fowler (FOWLER, 2003), “(…) o arquiteto deve ser como um guia (…) que é um experiente e capacitado membro da equipe que ensina aos outros se virar melhor, e ainda assim está sempre lá para as partes mais complicadas”. Larman (LARMAN, 2010) também diz que “um arquiteto que não está a par da evolução do código do produto, não está conectado com a realidade”.

3.2. Modelos Arquiteturais

3.2.1. Pipes e Filtros

O estilo arquitetural de pipes e filtros geralmente considera a existência de uma rede pela qual flui dados de uma extremidade (origem) à outra (destino). O fluxo de dados se dá através de *pipes* e os dados sofrem transformações quando processados nos filtros. Um *pipe* provê uma forma unidirecional de fluxo de dados, uma vez que atua como um condutor para o fluxo de dados entre a fonte (extremidade de origem dos dados) até um destino (receptor de dados). (MENDES, 2002).

Pipes e Filtros é um tipo de arquitetura orientada a dados, na qual os programas — chamados de filtros — têm como função processar os dados recebidos na entrada e gerar uma nova saída. Os filtros são conectados por meio de *pipes*, que agem como *buffers*. Isto é, *pipes* são usados para armazenar a saída de um filtro, enquanto ela não é lida pelo próximo filtro da sequência. Com isso, os filtros não precisam conhecer seus antecessores e sucessores, o que torna esse tipo de arquitetura bastante flexível, permitindo as mais variadas combinações de programas. Além disso, por construção, filtros podem ser executados em paralelo. (VALENTE, 2023).

O exemplo mais comum conhecido do estilo *pipes* e filtros é utilizado no sistema operacional Unix, por exemplo, a linha de comando: *cat* | *grep Brasil* | *sort* que especifica a execução de três comandos (filtros) que são conectados por dois *pipes* (barras verticais). No caso dos comandos Unix, as entradas e saídas são sempre arquivos texto. (VALENTE, 2023).

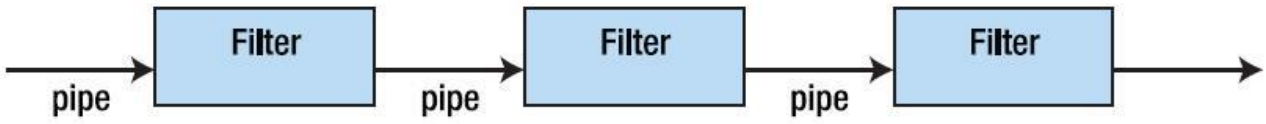

Figura 1 - Exemplo de uma arquitetura pipe e filtro

Fonte: https://www.devmedia.com.br/padrao-pipe-and-filter/29319

Para exemplificar melhor, foi criado um arquivo *.csv* com 6 colunas e executado o comando `cat | grep Brasil | sort`.

```
307,Valtteri Bottas,Alemanha,26,ForceIndia,Mercedes
160,Sebastian Vettel,Alemanha,26,Ferrari,Ferrari
167,Godofredo das Cove,Russia,30,Toro Rosso,Renault
128,Carlos Pepino,Ucrania,27,Mercedes,Mercedes
170,Nico Rosberg,Alemanha,25,Mercedes,Mercedes
136,Bruno Senna,Brasil,21,Hispanica,Cosworth
328,Orozimbo Nunba,Costa do Marfin,27,Williams,Mercedes
314,Nico Hulkenberg,Alemanha,33,ForceIndia,Mercedes
128,Nick Heidfeld,Alemanha,22,Sauber,Ferrari
235,Max Verstappen,Holanda,24,Toro Rosso,Renault
110,Lewis Hamilton,Reino Unido,27,Mercedes,Mercedes
112,Fernando Alonso,Espanha,28,Ferrari,Ferrari
210,Felipe Massa,Brasil,27,Williams,Mercedes
120,Rubens Barrichello,Brasil,29,Williams,Cosworth
```

Figura 2 - Arquivo exemplo-pipe.csv

Fonte: autoria própria

O comando *cat* faz a leitura do arquivo *csv*; o comando grep filtra as linhas que contenham a palavra Brasil e o *sort* classifica as linhas pela primeira coluna. O resultado do comando executado no terminal é apresentado na Figura 3.

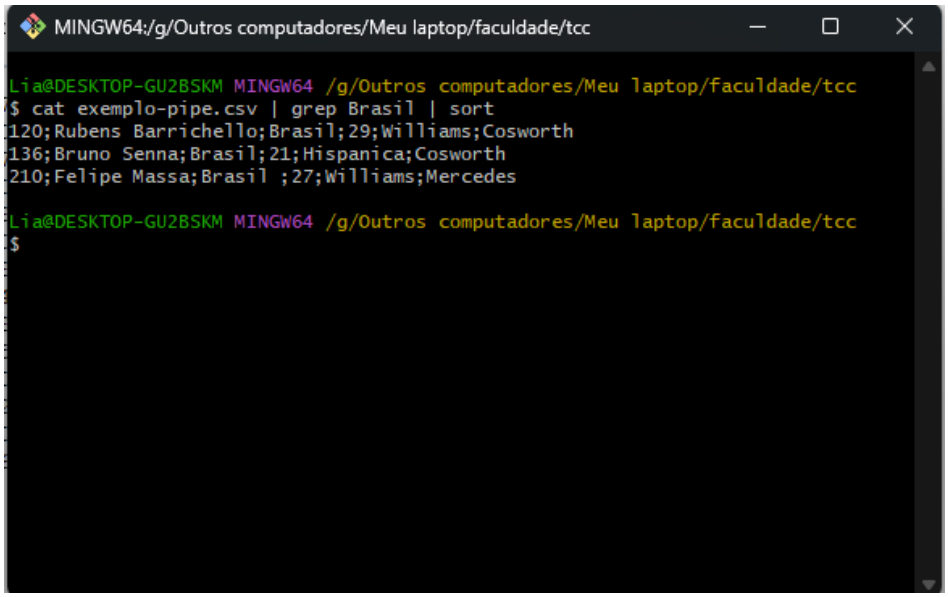

Figura 3 - Execução em pipes

Fonte: autoria própria

3.2.2. Camadas

O estilo arquitetural de camadas estrutura um sistema num conjunto de camadas, onde cada uma delas agrupa um conjunto de tarefas num determinado nível de abstração. Geralmente, numa arquitetura em camadas, uma camada num nível N oferece um conjunto de serviços à camada no nível superior, N+1. Para tanto, a camada N utiliza suas funções bem como faz uso dos serviços disponíveis na camada inferior, N-1. (MENDES, 2002).

Mendes (MENDES, 2002) diz que o melhor exemplo de uma arquitetura em camadas é o modelo OSI (Reference *Model for Open Systems Interconnection*) da ISO (*International Organization for Standization*). Nesse modelo, cada camada pode ser vista como um componente que pode ser implementado por software ou hardware. O modelo OSI da ISO possui sete camadas e serve como arquitetura de protocolos de redes de computadores.

Figura 4 - Arquitetura de camadas do modelo OSI da ISSO

Fonte: https://www.uniaogeek.com.br/arquitetura-de-redes-tcpip/

O número de camadas dependerá da funcionalidade a ser oferecida pelo sistema. Assim, faz-se necessário definir critérios de abstração a serem observados quando desejar agrupar subtarefas para comporem uma camada. Contudo, essa variação na arquitetura de camadas pode comprometer a manutenibilidade de um sistema. Haver um maior grau de dependência entre as camadas implica que mais de uma camada necessitará ser modificada para atender à mudança de requisitos. Outra observação é que, uma vez permitindo uma definição de vários níveis de abstração (camadas), ele proporciona uma maior flexibilidade ao sistema. Todavia isso tem um custo, o desempenho da arquitetura de camadas fica comprometido face à necessidade de uma solicitação externa ao sistema precisar passar por várias camadas a fim de ser tratada. (MENDES, 2002).

3.2.3. Objetos

A ideia fundamental por trás da abordagem orientada a objetos é combinar numa única entidade tanto os dados quanto as funções que atuam sobre esses dados. Essa entidade é denominada *objeto*. A abordagem orientada a objetos permite a organização de programas ao mesmo tempo em que ajuda a manter a integridade dos dados do sistema. Num projeto orientado a objetos, os projetistas de sistema podem encapsular dados e comportamento em objetos os quais possuem interfaces com

outros objetos. (MENDES, 2002).

Esse estilo possui várias propriedades importantes, como:

4. Objetos são entidades independentes (um tipo de dado abstrato) que podem sofrer modificações uma vez que toda informação pertinente é mantida no próprio objeto;
5. É possível fazer o mapeamento entre as entidades reais e objetos que atuam no controle de um sistema, resultando numa melhor compreensão do sistema;
6. Os objetos fazem uso de um mecanismo de troca de mensagens para se comunicar em vez de utilizar variáveis compartilhadas;
7. Objetos podem ser reutilizados devido à sua independência;
8. Os objetos podem estar distribuídos e executar sequencialmente ou em paralelo, a depender das decisões tomadas no início do projeto;

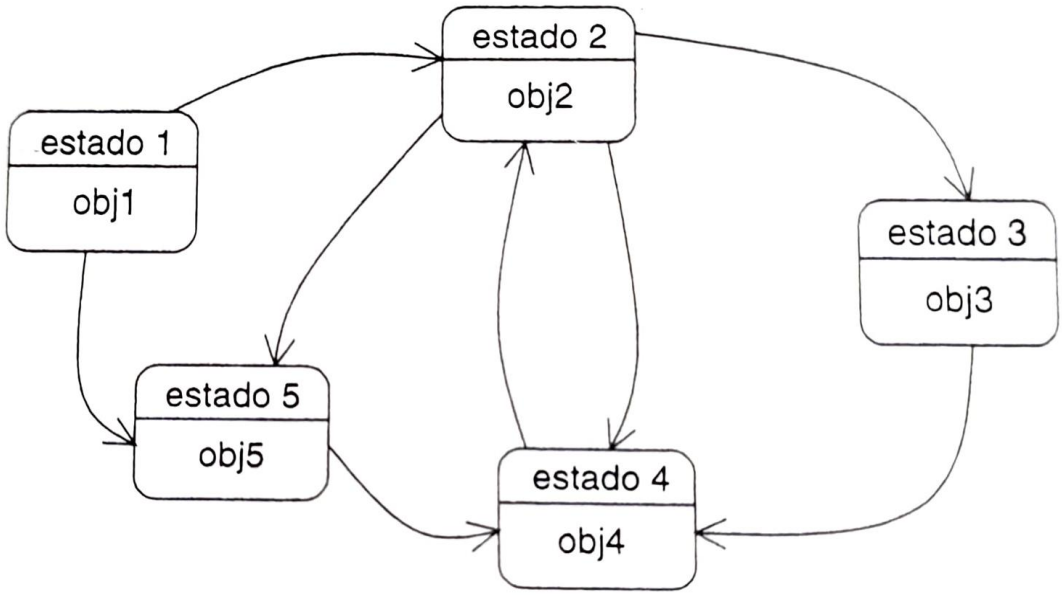

Figura 5 - Arquitetura de um projeto orientado a objetos

Fonte: https://www.devmedia.com.br/arquitetura-de-software-desenvolvimento-orientado-para-arquitetura/8033

A abordagem orientada a objetos tem como base a ocultação de informações, que é uma estratégia de projeto onde informações são escondidas em componentes do projeto. O estilo arquitetural de objetos vê um sistema de software como um conjunto de objetos comunicantes com estado associado a eles. Nesse caso, quando um objeto necessita se comunicar com outro objeto, ele precisa conhecer a identidade do objeto ao qual enviará uma mensagem. Dessa forma, numa arquitetura de objetos, teríamos um conjunto de objetos com estados próprios (escondidos) e as operações

associadas aqueles estados. Em tal arquitetura, os objetos (comunicantes) podem requisitar ou oferecer serviços a outros objetos. (MENDES, 2002).

Para que um objeto se comunicar com um outro objeto, a identidade do segundo precisa ser conhecida pelo primeiro e isso constitui numa desvantagem para o reuso pois terá que alterar a especificação de classes quando houver mudanças. (MENDES, 2002).

3.2.4. Quadro-Negro

O estilo quadro-negro ou blackboard originou-se no campo de inteligência artificial, no qual era utilizado como mecanismo para o compartilhamento de conhecimento ou dados por vários componentes inteligentes (NII, 1986).

A ideia do estilo arquitetural quadro-negro tem como base um modelo de solução de problema que fornece uma estrutura conceitual para organizar o conhecimento do domínio bem como uma estratégia para aplicar esse conhecimento. Esse estilo arquitetural prescreve a organização do conhecimento do domínio, todas as entradas e soluções parciais para solucionar o problema. (MENDES, 2002).

A arquitetura quadro-negro consiste basicamente de três componentes:

1. Células de conhecimento: o conhecimento necessário para solucionar um problema é particionado em células de conhecimento. Tais células são separadas e independentes;
2. Estrutura de dados do quadro-negro: os dados de solução de problemas são mantidos num banco de dados compartilhados, denominada de quadro-negro. As células de conhecimento causam modificações no quadro-negro que desencadeiam alterações até chegar numa solução. Toda interação e comunicação entre as células de conhecimento ocorrem unicamente através do quadro-negro;
3. Controle: as células de conhecimento respondem de forma oportunista às mudanças no quadro-negro;

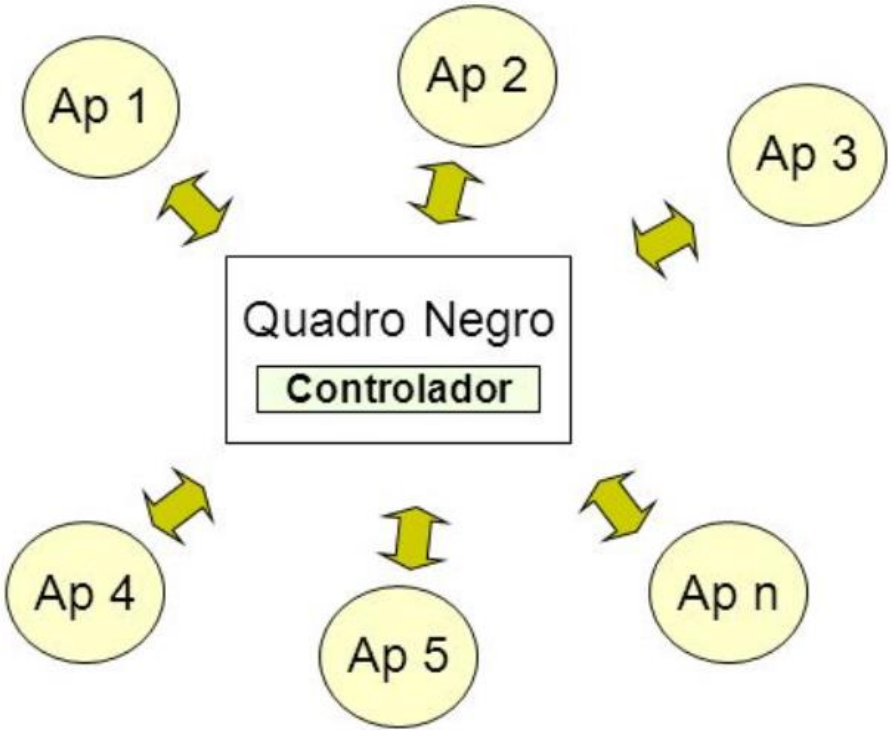

Figura 6 - Organização do quadro-negro

Fonte: Mendes, 2002

O estilo arquitetural quadro-negro possui um banco de dados compartilhado (controlador) que possui uma organização dependente da aplicação. Existe também, um conjunto de células de conhecimento que são logicamente independentes e respondem às mudanças que ocorrem no quadro-negro. As células de conhecimento podem ser ativadas através do estado do quadro-negro. Dessa forma, as células de conhecimento respondem às alterações que acontecem no quadro-negro. (MENDES, 2002).

Esse tipo de arquitetura é adequado em aplicações, onde diversos tipos de conhecimento devem ser considerados a fim de dar suporte à interpretação de um conjunto de dados iniciais. Tipicamente, a arquitetura quadro-negro era utilizada em casos onde não havia soluções gerais para um problema. Nesse caso, um ou mais componentes (células do conhecimento) interagem com o banco de dados compartilhado (quadro-negro) buscando encontrar uma solução parcial ou total. (MENDES, 2002).

A arquitetura quadro-negro mais simples envolve apenas um único repositório central de dados (quadro-negro). Entretanto, problemas mais complexos podem exigir múltiplos repositórios organizados num sistema distribuído de acordo com os tipos de

dados. A necessidade de um componente (célula do conhecimento) ‘passear’ pelo repositório de dados (quadro-negro) em busca de uma solução constitui-se numa desvantagem em termos de desempenho para esse estilo arquitetural. (MENDES, 2002).

De acordo com Mendes (MENDES, 2002), um aspecto positivo desse estilo arquitetural é a facilidade na qual consegue-se adicionar ou remover quantidade e tipos de dados. Uma vez que o processamento de cada componente é independente dos demais, componentes podem ser adicionados ou removidos sem acarretar modificações nos demais.

3.2.5. Cliente-Servidor

O estilo arquitetural cliente-servidor permite que as tarefas sejam divididas entre produtores e consumidores de dados. Um servidor é um processo que fica num estado de espera, aguardando solicitação de serviço de um ou mais clientes. (MENDES, 2002).

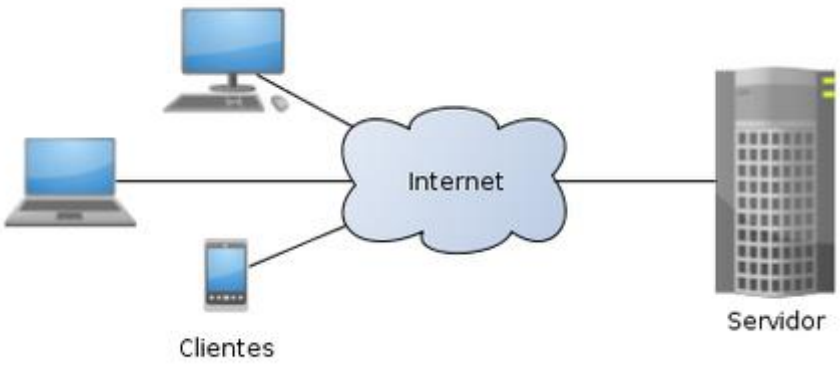

Figura 7 - Organização da arquitetura de cliente-servidor

Fonte: https://pt.wikipedia.org/wiki/Ficheiro:Cliente-Servidor.png

De acordo com Mendes (MENDES, 2002), um cenário típico envolvendo cliente e servidor seria:

1. Um sistema computacional acomoda um (processo) servidor, o qual é inicializado sozinho. Após inicializado, ele entra num estado de espera e aguarda que algum cliente entre em contato com ele, requisitando algum serviço;
2. Um (processo) cliente, que pode estar no mesmo sistema ou em algum

outro, é conectado ao servidor via rede. Geralmente, os clientes podem interagir com o usuário que digita algum comando. Assim, o cliente envia uma solicitação via rede ao servidor para que este último atenda a solicitação, possivelmente, efetuando alguma computação;

3. Após o servidor tratar a solicitação, ele retorna o resultado para o cliente, voltando em seguida para o estado de espera, no qual permanecerá até que chegue nova solicitação de algum cliente;

O servidor pode trabalhar de forma síncrona ou assíncrona. Quando operando no modo síncrono, o servidor retorna o controle ao cliente no mesmo instante em que ele envia os dados resultantes. Quando opera no modo assíncrono, ele retornará apenas os dados resultantes ao cliente ao término do tratamento da solicitação recebida. Os clientes no estilo arquitetural cliente-servidor podem ser vistos como processos que atuam de forma independente, isto é, a execução de um processo não interfere em outros. (MENDES, 2002).

Mendes (MENDES, 2002) discute que essa independência entre os processos implica nos seguintes benefícios para esse estilo:

- Facilidade de remover e/ou adicionar clientes. Essa facilidade de mudança decorre da independência existente entre os processos;
- Facilidade de modificar a funcionalidade de um cliente;

Porém Mendes (MENDES, 2002) também diz que tais benefícios podem não se verificar se houver acoplamento entre os clientes, o que acarreta maior nível de dependência entre os (processos) clientes.

3.2.6. Monolitíco

A arquitetura monolítica é um padrão de desenvolvimento de software no qual um aplicativo é criado com uma única base de código, um único sistema de compilação, um único binário executável e vários módulos para negócios ou recursos técnicos. Seus componentes trabalham juntos, compartilham o mesmo espaço de memória e recursos. (SAKOVICH, 2023).

Uma aplicação monolítica é aquele tipo de aplicação na qual toda a base de código está contida em um só lugar, ou seja, todas as funcionalidades estão definidas

no mesmo bloco. (SANTOS, 2023).

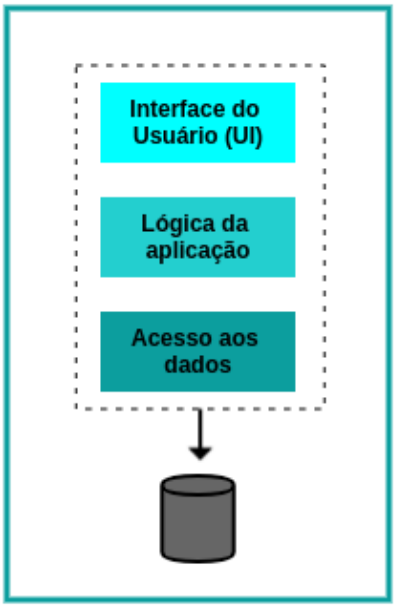

Figura 8 - Organização da arquitetura monolítica

Fonte: https://blog.4linux.com.br/microsservicos-o-que-sao-onde-habitam-e-como-a-4linux-pode-te-ajudar/

Geralmente este bloco é dividido em 3 partes:

- Apresentação: essencialmente é a camada de visualização, que será mostrada para o usuário.
- Negócio: é a camada que contém a lógica da aplicação. Nesta camada geralmente se encontram todas as bases de código, chamadas, *Application Programming Interface* (API's) e literalmente toda a inteligência do sistema;
- Dados: temos apenas as classes responsáveis pela conexão com o sistema de armazenamento de dados utilizado;

Por se tratar de um único sistema que reúne todas as aplicações e informações em um único banco de dados, o seu gerenciamento se torna mais simples e depende exclusivamente de uma única equipe. Porém, esse mesmo fator dificulta a escalabilidade do sistema, se houver um pico de demanda em uma determinada aplicação, todo o sistema terá que ser escalado. Outro ponto para se destacar na arquitetura monolítica é que ao longo do tempo a complexidade da atualização ou aplicação de um recurso do sistema se torna mais complicada, por conta da alta concentração no número de dados. (STAFF, 2023).

3.2.7. Hexagonal

Também chamada de *Ports and Adapters*, a arquitetura hexagonal é uma forma de organizar o código em camadas, cada qual com a sua responsabilidade, tendo como objetivo isolar totalmente a lógica da aplicação do mundo externo. Este isolamento é feito por meio de portas e adaptadores, onde as portas são as interfaces que as camadas de baixo nível expõe, e adaptadores as implementações para as interfaces em questão. (TRINDADE, 2023).

A ideia da arquitetura hexagonal é construir sistemas que favorecem reusabilidade de código, alta coesão, baixo acoplamento, independência de tecnologia e que são mais fáceis de serem testados. (VALENTE, 2023).

Uma arquitetura hexagonal divide as classes de um sistema em dois grupos principais:

4. Classes de domínio, isto é, diretamente relacionadas com o negócio do sistema;
5. Classes relacionadas com infraestrutura, tecnologias e responsáveis pela integração com sistemas externos;

Visualmente, a arquitetura é representada por meio de dois hexágonos concêntricos. No hexágono interno, ficam as classes do domínio (ou classes de negócio). No hexágono externo, ficam os adaptadores. Por fim, as classes de interface com o usuário, classes de tecnologia ou de sistemas externos ficam fora desses dois hexágonos. (VALENTE, 2023).

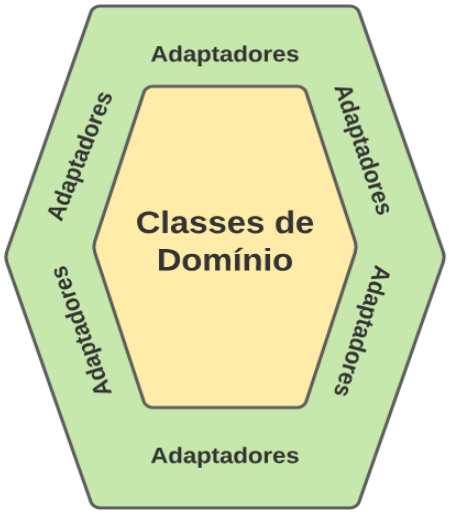

Figura 9 - Arquitetura Hexagonal

Fonte: https://engsoftmoderna.info/artigos/arquitetura-hexagonal.html#:~:text=Os%20objetivos%20de%20uma%20Arquitetura,mais%20f%C3%A1ceis%20de%20serem%20testados

A ideia central da arquitetura hexagonal é que o núcleo do sistema, que contém a lógica de negócio, deve ser independente das tecnologias externas, como bancos de dados, frameworks e interfaces de usuário. As portas são interfaces que definem os contratos entre o núcleo do sistema e os componentes externos. Elas representam os pontos de entrada e saída de dados do sistema, como APIs, interfaces gráficas e serviços externos. As portas permitem que o núcleo do sistema se comunique com o mundo externo de forma genérica, sem depender de implementações específicas. (Fenestra, 2023).

Já os adaptadores são responsáveis por implementar as portas, adaptando as chamadas e respostas do núcleo do sistema para as tecnologias externas. Eles são os responsáveis por lidar com detalhes técnicos, como a persistência de dados em um banco de dados específico ou a comunicação com um serviço externo. Os adaptadores garantem a separação entre o núcleo do sistema e as tecnologias externas, facilitando a substituição ou atualização dessas tecnologias no futuro. (Fenestra, 2023).

Com a arquitetura hexagonal, é possível testar a lógica de negócio de forma isolada, sem depender das implementações técnicas externas. Isso é possível através da criação de testes unitários que simulam as entradas e saídas do sistema, utilizando implementações de portas que são independentes das tecnologias reais. Além disso, a arquitetura hexagonal facilita a evolução do sistema ao longo do tempo. Como o núcleo do sistema é independente das tecnologias externas, é possível substituir ou atualizar essas tecnologias sem afetar a lógica de negócio. (Fenestra, 2023).

Porém esse tipo de arquitetura não é recomendado para pequenas aplicações, pois a complexidade adicionada na análise e separação dos componentes não deve trazer grandes ganhos. (TEMPORIN, 2023).

3.2.8. Microserviços

Uma arquitetura de microserviços consiste em uma coleção de pequenos serviços autônomos. Cada serviço é independente e deve implementar uma única funcionalidade comercial em um contexto limitado (MICROSOFT, 2023). Suas características são:

1. Os microserviços são pequenos, independentes e fracamente acoplados. Uma única equipe pequena de desenvolvedores pode escrever e manter um serviço.

2. Cada serviço é uma base de código separado, que pode ser gerenciado por uma equipe de desenvolvimento pequena.
3. Os serviços podem ser implantados de maneira independente. Uma equipe pode atualizar um serviço existente sem recompilar e reimplantar o aplicativo inteiro.
4. Os serviços são responsáveis por manter seus próprios dados ou o estado externo. Isso é diferente do modelo tradicional, em que uma camada de dados separada lida com a persistência de dados.
5. Os serviços comunicam-se entre si por meio de APIs bem definidas. Detalhes da implementação interna de cada serviço ficam ocultos de outros serviços.
6. Suporte à programação poliglota. Por exemplo, os serviços não precisam compartilhar a mesma pilha de tecnologia, bibliotecas ou estruturas.

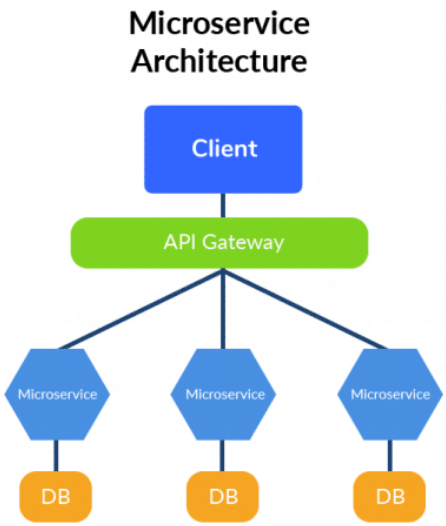

Figura 10 - Arquitetura de Microserviços

Fonte: https://cynoteck.com/pt/blog-post/microservices-vs-api/

Já que os microserviços são implantados de forma independente, é mais fácil gerenciar correções de bugs e lançamentos de recursos. Um microserviço deve ser pequeno o suficiente para que possa ser criado, testado e implantado por uma única equipe de recurso e tamanhos pequenos de equipe promovem maior agilidade. (MICROSOFT, 2023).

Em um aplicativo monolítico, há uma tendência de que, ao longo do tempo, as dependências de código fiquem entrelaçadas. A adição de um novo recurso requer a

alteração do código em muitos lugares. Não compartilhando código nem armazenamentos de dados, uma arquitetura de microserviços minimiza as dependências, o que torna mais fácil adicionar novos recursos. Se um microserviço individual ficar indisponível, ele não interromperá todo o aplicativo, desde que seja projetado para lidar com falhas corretamente. (MICROSOFT, 2023).

De acordo com o site da Microsoft (MICROSOFT, 2023), a arquitetura de microserviços possui alguns desafios a serem enfrentados, como por exemplo, cada serviço é mais simples, mas o sistema como um todo é mais complexo. Escrever um serviço pequeno que precisa de outros serviços dependentes exige uma abordagem diferente de escrever um aplicativo monolítico ou em camadas tradicionais. A arquitetura em microserviços pode acabar com muitos idiomas e estruturas diferentes que torna difícil a manutenção do sistema.

3.3. Caso de Estudo – Arquitetura Monolítica

De acordo com Fernandes (FERNANDES, 2020), monólito significa “obra construída em uma só pedra” e por isso é utilizado para definir a arquitetura de alguns sistemas e refere-se a forma de desenvolver um sistema, programa ou aplicação onde todas as funcionalidades e códigos estejam em um único processo. Essas diversas funcionalidades estão em um mesmo código fonte e em sua execução compartilham recursos da mesma máquina, seja processamento, memória, bancos de dados e arquivos.

A arquitetura monolítica é autossuficiente e independente de outras aplicações computacionais. O conceito dessa arquitetura é que o aplicativo não seja apenas responsável por uma determinada tarefa, mas também execute todas as etapas necessárias para completar uma macro função. (FERNANDES, 2020).

Na engenharia de software, um aplicativo monolítico descreve um aplicativo de software projetado sem modularidade externa, ou seja, um aplicativo projetado sem a necessidade de considerar a construção de módulos que possam se tornar componentes de outro aplicativo. Como o sistema está inteiro em um único bloco, seu desenvolvimento é mais ágil, se comparado com outras arquiteturas, sendo possível desenvolver uma aplicação em menos tempo e com menor complexidade inicial. (FERNANDES, 2020).

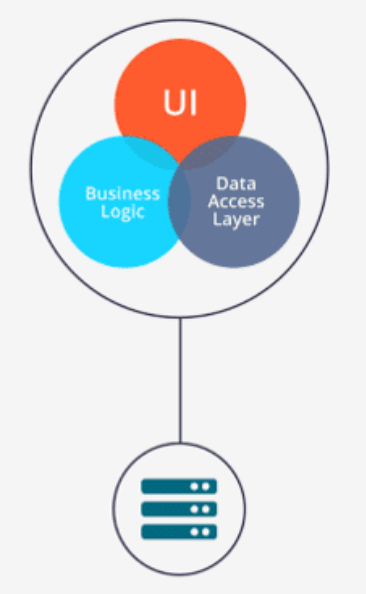

Figura 11 - Modelo de arquitetura monolítica

Fonte: https://www.nucleodoconhecimento.com.br/tecnologia/arquitetura-de-microsservicos

Para exemplificar o modelo de arquitetura monolítico, foi criado um sistema de agenda telefônica, uma aplicação desktop, utilizando a linguagem Java integrado com um banco de dados PostgreSQL local. Para criar a interface do sistema, foi utilizada a classe *Swing* do NetBeans, onde foi adicionado um *JFrame* para executar os serviços listar os contatos e adicionar um novo contato.

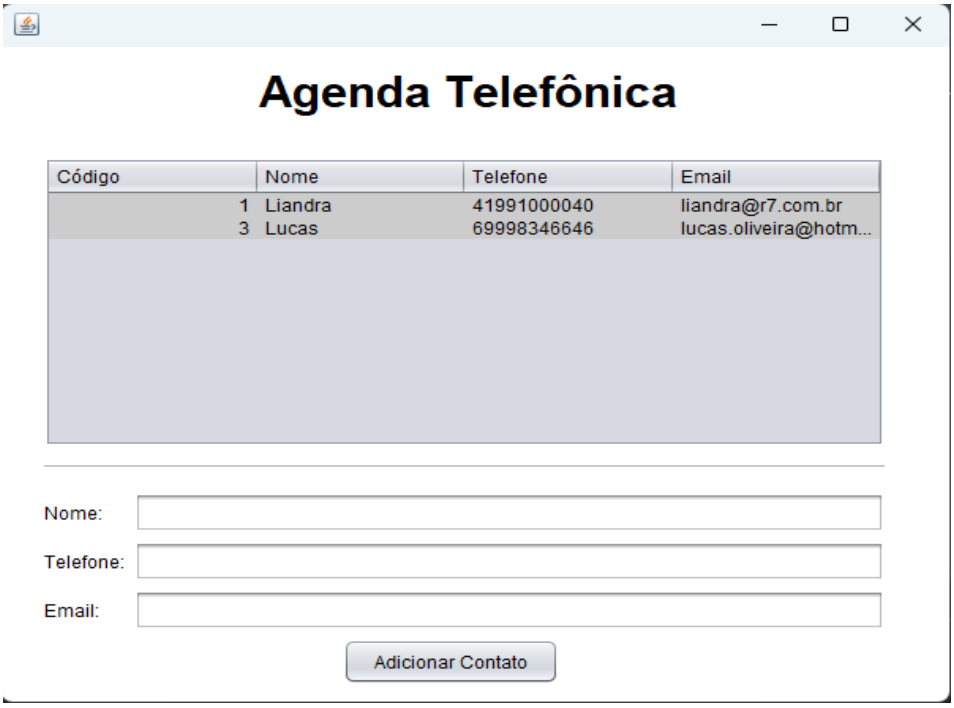

Figura 13 - Interface do sistema de agenda usando a arquitetura monolítica

Fonte: autoria própria

```
85  private void listarContatos() throws SQLException {
86      // Cria conexão com o banco de dados local
87      String url = "jdbc:postgresql://localhost:5432/projetotcc";
88      String usuario = "postgres";
89      String senha = "postgres";
90
91      Connection conexao = DriverManager.getConnection(url, user: usuario, password: senha);
92      PreparedStatement consulta = null;
93      ResultSet resultado = null;
94
95      try {
96
97          DefaultTableModel tabela = (DefaultTableModel) tabelaDeDados.getModel();
98          tabela.setRowCount(rowCount: 0);
99
100         // Cria consulta para trazer os dados do banco
101         String sql = "SELECT * FROM pessoa ORDER BY nome";
102         consulta = conexao.prepareStatement(string:sql);
103         resultado = consulta.executeQuery();
104
105         while (resultado.next()) {
106             Integer codigo = resultado.getInt(string:"codigo");
107             String nome = resultado.getString(string:"nome");
108             String telefone = resultado.getString(string:"telefone");
109             String email = resultado.getString(string:"email");
110
111             // Adiciona os dados na tabela para visualização
112             tabela.addRow(new Object[] {codigo, nome, telefone, email});
113         }
114
115
116     } catch (SQLException e) {
117         JOptionPane.showMessageDialog(
```

Figura 12 - Método listarContatos

Fonte: autoria própria

```java
private void adicionarContatoActionPerformed(java.awt.event.ActionEvent evt) {
    // Coleta os dados do contato da tela
    String nome = txtNome.getText();
    String telefone = txtTelefone.getText();
    String email = txtEmail.getText();

    // Aplicação das regras de negócio
    if (nome.isEmpty() || nome.isBlank()) {
        JOptionPane.showMessageDialog(
                parentComponent: this,
                message: "O campo nome é de preenchimento obrigatório.",
                title: "Erro    ",
                messageType: JOptionPane.ERROR_MESSAGE);

    } else if (telefone.isEmpty() || telefone.isBlank()) {
        JOptionPane.showMessageDialog(
                parentComponent: this,
                message: "O campo telefone é de preenchimento obrigatório.",
                title: "Erro    ",
                messageType: JOptionPane.ERROR_MESSAGE);

    } else {
        // Cria conexão com o banco de dados local
        String url = "jdbc:postgresql://localhost:5432/projetotcc";
        String usuario = "postgres";
        String senha = "postgres";

        try (Connection conexao = DriverManager.getConnection(url, user: usuario, password: senha)) {
            PreparedStatement consulta = null;
            ResultSet resultado = null;
```

Figura 15 - Método para adicionar um contato

Fonte: autoria própria

```java
        try (Connection conexao = DriverManager.getConnection(url, user: usuario, password: senha)) {
            PreparedStatement consulta = null;
            ResultSet resultado = null;

            // Cria consulta para adicionar um novo contato no banco
            String sql = "INSERT INTO pessoa (nome, telefone, email) VALUES (?, ?, ?)";

            try (PreparedStatement preparedStatement = conexao.prepareStatement(string:sql)) {

                preparedStatement.setString(i: 1, string:nome);
                preparedStatement.setString(i: 2, string:telefone);
                preparedStatement.setString(i: 3, string:email);

                // Insere contato
                int linhasAfetadas = preparedStatement.executeUpdate();

                if (linhasAfetadas > 0) {
                    JOptionPane.showMessageDialog(
                            parentComponent: this,
                            message: "Dados inseridos com sucesso!",
                            title: "Sucesso",
                            messageType: JOptionPane.INFORMATION_MESSAGE);

                } else {
                    JOptionPane.showMessageDialog(
                            parentComponent: this,
                            message: "Falha ao inserir os dados.",
                            title: "Erro    ",
                            messageType: JOptionPane.ERROR_MESSAGE);
                }

                listarContatos();
```

Figura 14 - Método para adicionar um contato

Fonte: autoria própria

No método *listarContatos*, foi criada a conexão com o banco de dados local. Após a conexão bem sucedida, é feita a consulta para trazer os contatos, onde os dados são adicionados na tabela para a visualização do usuário, e a conexão com o banco é fechada.

No método *adicionarContatoActionPerformed*, primeiramente é coletado os dados preenchidos pelo usuário. Em seguida são aplicadas as regras de negócio, que validam as informações verificando se os campos Nome e Telefone são nulos. Caso os dados sejam válidos, novamente, é criado a conexão com o banco de dados local e a consulta para inserir os dados e a conexão com o banco é fechada e os contatos são listados em seguida.

No modelo de Agenda Telefônica, é possível identificar a arquitetura monolítica, onde todo o desenvolvimento do software é concentrado em uma única base de código. Neste exemplo criado, observa-se que a interface – onde é possível adicionar um novo contato no botão; a lógica de negócio – onde é verificado se os dados são válidos; e o banco de dados estão todos concentrados na ação do botão.

A vantagem deste modelo de arquitetura é sua simplicidade no desenvolvimento, onde toda a organização fica concentrada em um único sistema. No entanto, sua manutenção de torna complexa, pois uma vez que o sistema se torna maior, seu código ficará mais difícil de entender.

Além disso, o modelo monolítico não possui flexibilidade. Caso seja necessário mudar, por exemplo, para um banco de dados em nuvem ou alterar a linguagem do sistema, todo o código precisará ser alterado e não há reaproveitamento de código.

3.4. Caso de Estudo – Arquitetura em Camadas

Arquitetura em camadas é um estilo arquitetural que dentre vários objetivos, destaca-se o de organizar as responsabilidades de partes de um software, normalmente criando um isolamento e dando um propósito bem definido a cada camada de forma que a mesma possa ser reutilizável por um nível mais alto ou até substituível. (SOUZA, 2023)

Em sistemas que seguem esse padrão de arquitetura, as classes são organizadas em módulos de maior tamanho, chamados de camadas. As camadas são dispostas de forma hierárquica, como em um bolo, em que cada camada repousa sobre uma camada mais baixa. (VALENTE, 2022)

Para exemplificar o modelo de arquitetura em camadas, o sistema criado usando a arquitetura monolítica, foi evoluído para um modelo de arquitetura em camadas.

O exemplo desenvolvido utiliza 5 camadas: a parte do *frontend* foi dividida em duas camadas, apresentação e controladora, e a parte do *backend* possui as camadas de controle, de negócio e uma camada de acesso ao banco de dados.

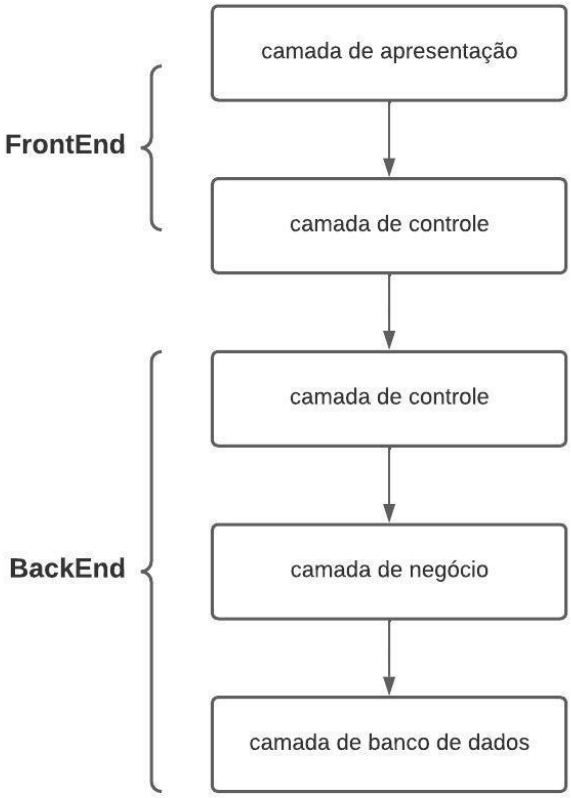

Figura 16 - Arquitetura do sistema de agenda telefônica

Fonte: autoria própria

Para o backend, esse modelo foi desenvolvido utilizando o Eclipse IDE, a linguagem Java, um banco de dados em nuvem, o Azure Cloud, e o framework Spring Boot, para facilitar o desenvolvimento de aplicações Web. Além disso o Spring Boot permite otimizar várias consultas usando Java Persistence API (JPA), fazer injeções de dependências e utilizar notações para validar dados. Para o *frontend*, o sistema foi desenvolvido utilizando Linguagem de Marcação de HiperTexto (HTML), Cascading Style Sheets (CSS), JavaScript e AngularJS.

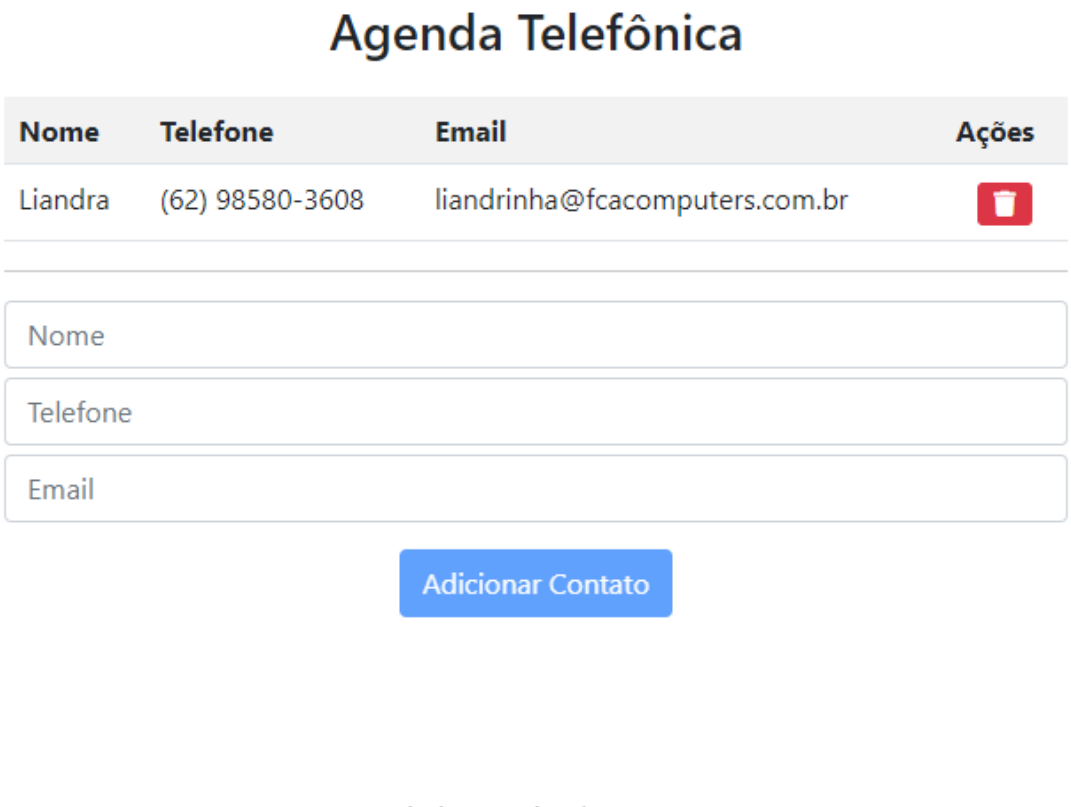

Figura 17 - Interface do sistema de agenda utilizando a arquitetura em camadas

Fonte: autoria própria

A parte do *frontend* foi dividida em 2 camadas, uma camada de apresentação e uma camada de controle. A camada de apresentação é responsável por toda a apresentação dos dados e pela interação com o usuário. A camada de controle, ficará responsável pela interação entre a camada de apresentação e a camada de negócio além de processar as informações que vem dessa camada, e permitir que a camada de apresentação seja dinâmica.

Na interface apresentada na Figura 17, é possível observar um conjunto de serviços – assim como no modelo de arquitetura monolítica; como a listagem de contatos, e os campos de Nome, Telefone e Email juntamente com um botão para adicionar um novo contato.

```html
<body ng-controller="AgendaTelefonicaController">
    <div class="jumbotron">
        <h3>{{app}}</h3>

        <table class="table table-striped">
            <tr>
                <th>Nome</th>
                <th>Telefone</th>
                <th>Email</th>
                <th>Ações</th>
            </tr>

            <tr ng-repeat="contato in contatos">
                <td>{{contato.nome}}</td>
                <td>{{contato.telefone | telefone}}</td>
                <td>{{contato.email}}</td>
                <td id="btnExcluir">
                    <button class="btn btn-danger btn-sm" ng-click="apagarContato(contato)">
                        <i class="fas fa-trash"></i>
                    </button>
                </td>
            </tr>
        </table>
        <hr/>

        <input class="form-control" type="text" ng-model="contato.nome" name="nome" placeholder="Nome" maxlength="255"/>
        <input class="form-control" type="text" ng-model="contato.telefone" name="telefone" placeholder="Telefone" ui-br-phone-number />
        <input class="form-control" type="text" ng-model="contato.email" name="email" placeholder="Email" maxlength="255"/>
        <button class="btn btn-primary btn-block" ng-click="adicionarContato(contato)" ng-disabled="!contato.nome || !contato.telefone || !contato.email">Adicionar Contato</button>
    </div>
</body>

<div style="text-align: center;" id="rodape">
    Criado por Liandra Lopes
</div>
</html>
```

Figura 19 - Camada de apresentação do frontend

Fonte: autoria própria

```javascript
angular.module("agendaTelefonica").controller("AgendaTelefonicaController", function ($scope, ContatosService) {
    $scope.app = 'Agenda Telefônica';

    function buscarContatos() {
        ContatosService.getContatos().then(function(response) {
            $scope.contatos = response.data;
        });
    }

    buscarContatos();

    $scope.adicionarContato = function (contato) {
        ContatosService.cadastrar(contato).then(function (response) {
            bootbox.alert('Contato salvo com sucesso!');
            buscarContatos();
            $scope.contato = null;
        }).catch( function (error) {
            var mensagemDeErro = error.data.message.join("</br>");
            bootbox.alert(mensagemDeErro);
        });
    };

    $scope.apagarContato = function (contato) {
        bootbox.confirm("Deseja deletar o contato? \n Essa ação não poderá ser alterada.",
        function(result) {
            if (result) {
                ContatosService.apagar(contato.codigo).then(function () {
                    bootbox.alert('Contato excluído com sucesso!');
                    buscarContatos();
                });
            }
        });
    }

});
```

Figura 18 - Camada de controle do frontend

Fonte: autoria própria

Na classe *PessoaDTO* – Figura 20, temos a camada de lógica de negócio, onde os dados informados pelo usuário serão validados utilizando anotações de validação do Spring Boot. No sistema de Agenda Telefônica, temos as seguintes regras de negócio para adicionar um contato: o nome e o telefone não podem estar nulos e o e-mail precisa ser um e-mail válido.

```
12  @JsonInclude(Include.NON_NULL)
13  @JsonIgnoreProperties(ignoreUnknown = true)
14  public class PessoaDTO implements Serializable {
15
16      private static final long serialVersionUID = -546768257650691313L;
17
18      private Integer codigo;
19
20      @NotNull(message = "O campo nome é de preenchimento obrigatório")
21      private String nome;
22
23      @NotNull(message = "O campo telefone é de preenchimento obrigatório")
24      private String telefone;
25
26      @Email(message = "Preencha o campo com um email válido.")
27      private String email;
28
29      public PessoaDTO() {}
30
```

Figura 20 - Camada de negócio do backend - classe PessoaDTO

Fonte: autoria própria

Na classe *PessoaService* – Figura 22, temos uma camada de controle, onde os dados são manipulados para serem enviados ou buscados da camada de banco de dados.

A classe *PessoaRepository* – Figura 21, é a camada de banco de dados, que irá armazenar os dados manipulados pelo sistema. No serviço para adicionar um contato por exemplo, após os dados serem validados, a função *cadastrar* será responsável por salvar os dados no banco de dados.

```java
@Service
public class PessoaService extends ConsultaPadraoServiceAbstract<PessoaDTO, Integer>{

    protected PessoaService(ConsultaPadraoRepositoryInterface<PessoaDTO, Integer> repository) {
        super(repository);
    }

    @Autowired
    private PessoaRepository pessoaRepository;

    @Transactional
    public void cadastrar(final PessoaDTO filtro) {
        this.pessoaRepository.cadastrar(preencheEntidade(filtro));
    }

    private Pessoa preencheEntidade(final PessoaDTO filtro) {
        final Pessoa entidade = new Pessoa();
        entidade.setNome(filtro.getNome());
        entidade.setTelefone(filtro.getTelefone());
        entidade.setEmail(filtro.getEmail());
        return entidade;
    }

    @Transactional
    public void excluirContato(final Integer codigoDaPessoa) {
        if (codigoDaPessoa != null) {
            this.pessoaRepository.excluirContato(codigoDaPessoa);
        }
    }
}
```

Figura 22 - Camada de controle do backend - classe PessoaService

Fonte: autoria própria

```java
@Repository
@Transactional(propagation = Propagation.MANDATORY)
public class PessoaRepository
    implements ConsultaPadraoRepositoryInterface<PessoaDTO, Integer> {

    @PersistenceContext
    protected EntityManager entityManager;

    protected CriteriaBuilder cb() {
        return this.entityManager.getCriteriaBuilder();
    }

    @Override
    public List<PessoaDTO> buscarTodos() {
        final CriteriaQuery<PessoaDTO> criteriaQuery = this.cb().createQuery(PessoaDTO.class);
        final Root<Pessoa> root = criteriaQuery.from(Pessoa.class);
        criteriaQuery.multiselect(montarSelectParaBuscarTodos(root));

        return this.entityManager.createQuery(criteriaQuery).getResultList();
    }

     private List<Selection<?>> montarSelectParaBuscarTodos(final Root<Pessoa> root) {
            return Arrays.asList(
                    root.get(Pessoa_.codigo),
                    root.get(Pessoa_.nome),
                    root.get(Pessoa_.telefone),
                    root.get(Pessoa_.email));
    }

    public void cadastrar(final Pessoa entidade) {
        this.entityManager.persist(entidade);
    }

}
```

Figura 21 - Camada de banco de dados – PessoaRepository

Fonte: autoria própria

No modelo apresentado acima, é possível concluir que existe um modelo em camadas aplicado. A camada de apresentação, irá tratar da interação do usuário com o sistema, como adicionar um contato por exemplo, e essa camada não irá conhecer a regra de negócio. A camada de negócio será responsável por validar os dados informados pelo usuário e a camada de persistência dos dados irá se conectar com o banco de dados e persistir os dados manipulados.

As vantagens desse modelo de arquitetura é que quando as camadas são bem definidas, elas podem garantir a reutilização de código, além de isolar diferentes aspectos do sistema, o que o torna mais organizado e mais fácil de entender e dar manutenção. Outra vantagem é poder modificar ou substituir uma camada sem afetar as demais, o que torna o sistema mais flexível.

Entretanto, em um modelo em camadas, a complexidade para gerenciar e fornecer manutenção em todas as camadas, à medida que o sistema cresce, pode aumentar bastante.

3.5. Caso de Estudo – Arquitetura de Microserviços

O microsserviço é uma arquitetura e uma abordagem para escrever programas de software. Com eles, as aplicações são desmembradas em componentes mínimos e independentes. Diferentemente da abordagem tradicional monolítica em que toda a aplicação é criada como um único bloco, os microsserviços são componentes separados que trabalham juntos para realizar as mesmas tarefas. Cada um dos componentes ou processos é um microsserviço. Essa abordagem de desenvolvimento de software valoriza a granularidade, a leveza e a capacidade de compartilhar processos semelhantes entre várias aplicações. (REDHAT, 2023).

Para demostrar o padrão de arquitetura em microserviços, foi utilizado o mesmo sistema de Agenda Telefônica e os mesmos serviços. Para esse modelo, foi utilizado o Eclipse IDE, a linguagem Java, Spring Boot Framework e banco de dados PostgreSQL em nuvem da Azure Cloud, o mesmo usado no modelo em camadas. O *frontend* foi mantido o mesmo do modelo em camadas.

Na arquitetura em microserviços, os serviços foram separados em serviços totalmente independentes. Sendo assim, foram criados 3 microserviços diferentes, um para listar contatos, outro para adicionar e outro para excluir um contato, onde cada

microserviço roda em portas diferentes.

No microserviço para listar os contatos, foi necessário criar duas classes, além da classe *Main* e uma interface. A classe *PessoaAPI* – Figura 23, ficará responsável retornar os contatos ao *frontend*, através de um método HTTP. Essa classe, terá uma injeção de dependência da interface *IPessoaRepository* – Figura 24, que irá estender a interface do JPA e ficará responsável pelas consultas e conexão com o banco de dados. A última classe é a entidade *Pessoa* – Figura 25, onde o JPA irá estabelecer uma ligação entre essa entidade e a tabela no banco de dados com o mesmo nome, onde esses dados do tipo Pessoa poderão ser persistidos.

```
10  @RestController
11  @RequestMapping("/api/")
12  @CrossOrigin(origins = "http://127.0.0.1:5500")
13  public class PessoaAPI {
14
15      private final IPessoaRepository pessoaRepository;
16
17      public PessoaAPI(IPessoaRepository pessoaRepository) {
18          this.pessoaRepository = pessoaRepository;
19      }
20
21      @GetMapping(value = "contatos")
22      public List<Pessoa> listarClientes() {
23          return pessoaRepository.findAll();
24      }
25
26  }
27
```

Figura 23 - Classe PessoaAPI no microserviço para listar contatos

Fonte: autoria própria

```
3  import org.springframework.data.jpa.repository.JpaRepository;
4
5  public interface IPessoaRepository extends JpaRepository<Pessoa, Integer> {
6
7  }
8
```

Figura 24 - Classe IPessoaRepository no microserviço para listar contatos

Fonte: autoria própria

```java
@Entity
@Table(name = "pessoa")
public class Pessoa {

    @Id
    @SequenceGenerator(name = "seq_pessoa_codigo", sequenceName = "seq_pessoa_codigo", allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_pessoa")
    @Column(name = "codigo", unique = true, nullable = false, updatable = false)
    private Integer codigo;

    @Column(name = "nome", nullable = false, length = 255)
    private String nome;

    @Column(name = "telefone", nullable = false, length = 11)
    private String telefone;

    @Column(name = "email", nullable = true, length = 255)
    private String email;

    public Pessoa() {}

    public Integer getCodigo() {
        return codigo;
    }
```

Figura 25 - Entidade Pessoa no microserviço para listar contatos

Fonte: autoria própria

No microserviço para adicionar um contato, a interface *IPessoaRepository* – Figura 23, se manteve a mesma. Na entidade *Pessoa* – Figura 26, foram adicionadas algumas notações para validar os dados inseridos pelo usuário e aplicar a regra de negócio. Já na classe *PessoaAPI* – Figura 27, foi criado um método HTTP para cadastrar um novo contato.

```java
@Entity
@Table(name = "pessoa")
public class Pessoa {

    @Id
    @SequenceGenerator(name = "seq_pessoa_codigo", sequenceName = "seq_pessoa_codigo", allocationSize = 1)
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "seq_pessoa")
    @Column(name = "codigo", unique = true, nullable = false, updatable = false)
    private Integer codigo;

    @Column(name = "nome", nullable = false, length = 255)
    @NotNull(message = "O campo nome é de preenchimento obrigatório")
    private String nome;

    @Column(name = "telefone", nullable = false, length = 11)
    @NotNull(message = "O campo telefone é de preenchimento obrigatório")
    private String telefone;

    @Column(name = "email", nullable = true, length = 255)
    @Email(message = "Preencha o campo com um email válido.")
    private String email;

    public Pessoa() {}
```

Figura 26 - Entidade Pessoa no microserviço para adicionar um contato

Fonte: autoria própria

```java
@RestController
@RequestMapping("/api/")
@CrossOrigin(origins = "http://127.0.0.1:5500")
public class PessoaAPI {

    private final IPessoaRepository pessoaRepository;

    public PessoaAPI(IPessoaRepository pessoaRepository) {
        this.pessoaRepository = pessoaRepository;
    }

    @PostMapping(value = "cadastrar")
    public void cadastrar(@Valid @RequestBody final Pessoa filtro, HttpServletResponse response){
        this.pessoaRepository.save(filtro);
    }

}
```

Figura 27 - Classe PessoaAPI no microserviço para adicionar um contato

Fonte: autoria própria

No microserviço para excluir um contato, a entidade *Pessoa* e a interface *IPessoaRepository* se mantiveram as mesmas do microserviço para listar um contato. A única classe alterada será a classe *PessoaAPI* – Figura 28 que irá retornar um método HTTP para excluir um contato.

```java
@RestController
@RequestMapping("/api/")
@CrossOrigin(origins = "http://127.0.0.1:5500")
public class PessoaAPI {

    private final IPessoaRepository pessoaRepository;

    public PessoaAPI(IPessoaRepository pessoaRepository) {
        this.pessoaRepository = pessoaRepository;
    }

    @DeleteMapping(value = "{codigo}")
    public ResponseEntity<Void> excluirContato(@PathVariable("codigo") final Integer codigoDaPessoa) {
        Optional<Pessoa> pessoa = pessoaRepository.findById(codigoDaPessoa);
        if (pessoa.isPresent()) {
            pessoaRepository.delete(pessoa.get());
            return ResponseEntity.noContent().build();
        } else {
            return ResponseEntity.notFound().build();
        }
    }

}
```

Figura 28 - Classe PessoaAPI no microserviço para excluir um contato

Fonte: autoria própria

Neste modelo utilizando a arquitetura em microserviços, nota-se que cada serviço se tornou independente, autônomo e fracamente acoplado. Isso significa que se algum desses serviços não funcionar corretamente, isso não terá impacto no funcionamento de nenhum dos outros serviços. Além disso, esse tipo de arquitetura

permite uma programação poliglota, já que não precisam compartilhar a mesma tecnologia, estrutura ou bibliotecas.

Entretanto, um sistema formado por várias partes autônomas aumenta a complexidade de gerenciamento e manutenção, visto que podem utilizar várias linguagens ou estruturas diferentes.

3.6. Ferramental Utilizado

- NetBeans IDE

O NetBeans foi inicialmente desenvolvido pela empresa checa NetBeans, mas foi adquirida pela Sun Microsystems em 1999. Em junho de 2000, o NetBeans tornou-se código aberto, que permaneceu como patrocinadora do projeto até janeiro de 2010, quando a Sun Microsystems se tornou uma subsidiária da Oracle. O NetBeans é um ambiente de desenvolvimento integrado (IDE) e oferece suporte integrado para várias linguagens de programação. Essa IDE foi escolhida para criar o modelo de arquitetura monolítico, pois fornece ferramentas gráficas para a criação de GUI (Interface Gráfica do Utilizador).

- PostgreSQL

O PostgreSQL é um sistema de gerenciamento de banco de dados relacional de código aberto que usa e estende a linguagem SQL combinada com muitos recursos que armazenam e dimensionam com segurança as cargas de trabalho de dados mais complicadas. As origens do PostgreSQL remontam a 1986 como parte do projeto POSTGRES da Universidade da Califórnia em Berkeley. O PostgreSQL foi escolhido como banco de dados das três aplicações criadas nos casos de uso pois ele garante a integridade dos dados mesmo em situações de falha. Além disso o PostgreSQL permite criar funções próprias, operadores, tipos de dados e índices e isso facilita a personalização e adaptação do sistema às necessidades específicas da aplicação.

- Eclipse IDE

Utilizado o Eclipse para codificação do *backend* dos modelos em camadas e de microserviços, o Projeto Eclipse foi originalmente criado pela IBM em novembro de 2001 como um projeto de código aberto. Em 2004, a Eclipse Foundation foi criada

como uma organização independente sem fins lucrativos para gerenciar o desenvolvimento contínuo do Eclipse. A fundação é apoiada por várias empresas de tecnologia e comunidades de desenvolvedores. O Eclipse foi escolhido pela sua facilidade de configurar o ambiente de desenvolvimento em Java, já que a IDE possui uma Java Virtual Machine (JVM) integrada. Também possui uma vasta gama de plugins disponíveis e suporte ao framework do Spring Boot. Além disso, o Eclipse possui uma curva de aprendizagem alta.

- Visual Studio Code

O Visual Studio Code foi anunciado pela Microsoft em abril de 2015 e lançado oficialmente em novembro do mesmo ano. O objetivo do Visual Code era oferecer uma ferramenta leve, rápida e altamente extensível que pudesse ser utilizada por desenvolvedores em várias plataformas e para uma ampla gama de linguagens de programação. O Visual Studio Code (VS Code), utilizado para programar a parte do *frontend* nos modelos em camadas e de microserviços, é um editor de código de código aberto desenvolvido pela Microsoft. O Visual Code foi escolhido pela leveza e por ter uma interface mais amigável, já que possui uma grande variedade de extensões disponíveis para personalizar e estender as funcionalidades, o que acelera o processo de desenvolvimento.

- Java 19

O Java é uma linguagem de programação que se iniciou em 1991, quando uma equipe da Sun Microsystems começou um projeto de uma linguagem com foco em orientação de projetos. Em 2009, a Oracle Corporation adquiriu a Sun Microsystems, tornando-se responsável pelo desenvolvimento e manutenção do Java.

A linguagem Java foi selecionada por ser uma linguagem orientada a objetos, então permite uma modelagem mais próxima dos conceitos do mundo real e também pelo fato de aplicações Java serem executadas em várias plataformas.

- Spring Boot

O Spring Boot é um projeto do Spring Framework que simplifica significativamente o desenvolvimento de aplicações Java. O projeto Spring Boot foi iniciado pela Pivotal Software, uma subsidiária da VMware, e a primeira versão foi lançada em 2014. A escolha do Spring Boot foi feita pela facilidade em criar aplicações

Java e também de gerenciar dependências, utilizando o *Spring Initializr*. Além disso, o Spring tem um mecanismo robusto para validação de dados, o que ajuda a otimizar código e torna as aplicações mais simples.

- HTML

HTML é a sigla para *HyperText Markup Language* (Linguagem de Marcação de Hipertexto, em inglês). Foi criado na década de 1990 pelo físico britânico Tim Berners-Lee, considerado o “pai da web”, para auxiliar na formatação dos documentos de pesquisas compartilhados entre ele e seus colegas. Foi escolhido pois o HTML é fundamental para a criação de páginas web e é uma linguagem de marcação simples e fácil de aprender. Além disso, é um padrão aberto, amplamente aceito e suportado por todos os principais navegadores.

- JavaScript

JavaScript é uma linguagem de programação de alto nível, dinâmica e orientada a objetos, amplamente utilizada para o desenvolvimento de aplicações web. JavaScript foi criado por Brendan Eich enquanto trabalhava na Netscape Communications Corporation. A linguagem foi originalmente chamada de "Mocha" e, posteriormente, de "LiveScript" antes de ser renomeada para JavaScript quando a Netscape fez uma parceria com a Sun Microsystems. Foi utilizado pois é leve, permite a criação de páginas dinâmicas, é suportado por todos os navegadores mais recentes, além de ser relativamente fácil de aprender.

- AngularJS

O AngularJS é um framework JavaScript de código aberto mantido pelo Google, desde 2010 projetado para facilitar o desenvolvimento de aplicações web de única página (SPA - Single Page Applications). Esse framework foi escolhido para o *frontend* pois permite utilizar o HTML como linguagem de modelo e estender a sintaxe do HTML para apresentar os componentes da aplicação de forma clara e sucinta. A ligação de dados e a injeção de dependências do AngularJS eliminam grande parte do código que, de outra forma, teria de escrever.

- Microsoft Azure Cloud

O Microsoft Azure foi lançado em fevereiro de 2010 como uma plataforma de

nuvem aberta e flexível. O Azure foi originalmente chamado de "Windows Azure" e fazia parte da visão da Microsoft de fornecer uma plataforma de nuvem abrangente para empresas e desenvolvedores. O Microsoft Azure é uma plataforma de computação em nuvem da Microsoft que oferece uma ampla gama de serviços e soluções para hospedar, gerenciar e desenvolver aplicativos em ambientes de nuvem. A escolha do Azure Cloud se deu pela variedade de serviços e banco de dados que são oferecidos, inclusive um banco de dados em nuvem em PostgreSQL.

4. CONCLUSÃO

Em suma, a importância da arquitetura de software é fundamental para o desenvolvimento bem-sucedido de sistemas e afeta diretamente a qualidade, o desempenho e a manutenção do software. O uso de modelos arquiteturais desempenha um papel crucial no fornecimento de uma estrutura abstrata que orienta o processo de construção. Ao escolher o estilo arquitetônico certo, levando em consideração fatores como tamanho, complexidade, orçamento e escalabilidade, os desenvolvedores podem não apenas simplificar o desenvolvimento do código, mas também melhorar a usabilidade e a eficiência do sistema.

O modelo monolítico, com sua abordagem integrada, simplifica o desenvolvimento e a manutenção, oferecendo uma visão consolidada do sistema. No entanto, pode enfrentar desafios em termos de escalabilidade e flexibilidade para sistemas complexos, visto que as atualizações e modificações podem impactar a totalidade do aplicativo.

Já o modelo em camadas proporciona uma divisão clara das responsabilidades, promovendo a modularidade e a reutilização de componentes. Essa abordagem facilita a escalabilidade e a manutenção, tornando-o ideal para sistemas que exigem separação de preocupações. No entanto, a comunicação entre camadas pode resultar em sobrecarga, afetando o desempenho em alguns casos.

Por fim, o modelo de microserviços, ao adotar uma abordagem descentralizada, permite a construção de sistemas altamente escaláveis e flexíveis. Cada microserviço opera de forma independente, facilitando atualizações e manutenção contínua. No entanto, a complexidade da gestão de múltiplos serviços e a necessidade de uma infraestrutura robusta podem representar desafios.

Em conclusão, a escolha entre esses modelos arquiteturais deve ser cuidadosamente ponderada, considerando as características específicas do projeto. A flexibilidade e a escalabilidade oferecidas pelos modelos em camadas e de microserviços podem ser vitais em ambientes dinâmicos, enquanto o modelo monolítico pode ser mais adequado para projetos menores e menos complexos. Em última análise, a escolha e aplicação cuidadosa da arquitetura de software e seus modelos são a chave para atingir resultados sólidos e sustentáveis na criação de sistemas de software robustos e eficazes.

5. REFERÊNCIAS BIBLIOGRÁFICAS

AngularJS. Disponível em: <https://docs.angularjs.org/guide/introduction#:~:text=AngularJS%20is%20a%20structural%20framework,application's%20components%20clearly%20and%20succinctly.>. Acesso em: 3 nov. 2023.

APACHE NETBEANS. **About Apache NetBeans**. Disponível em: <https://netbeans.apache.org/front/main/about/index.html>. Acesso em: 3 nov. 2023.

O que é arquitetura hexagonal. Fenestra, 2023. Disponível em: <https://fenestra.com.br/blog/2023/07/01/o-que-e-arquitetura-hexagonal/>. Acesso em: 13 set. 2023.

FERNANDES, Henrique Marques. **O que é um sistema/aplicação Monolito/Monolítica?**. Disponível em: <https://dev.to/shadowlik/o-que-e-um-sistema-aplicacao-monolito-monolitica-3o14>. Acesso em: 30 out. 2023.

FOWLER, Martin. **Who needs an architect?** In: IEEE Software, v. 20. 2003.

GARLAN, David; PERRY, Dewayne. **Introduction to the Special Issue on Software Architecture** IEEE Transactions on Software Engineering. Abril, 1995. p. 269-274.

International Organization for Standardization (International Eletrotechnical Committe. **Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 1: Basic Reference Model**. International Standard 7498-1, 1984

International Organization for Standardization (International Eletrotechnical Committe. **Information Technology – Telecommunications and Information Exchange Between Systems – Overview of Local Area Network Standards**. Draft Technical Report, 8802-1.2, 1994

JULIA. **Java - Vantagens e Desvantagens ⋆ Conteige Cloud**. Disponível em: <https://conteige.cloud/java-vantagens-e-desvantagens/>. Acesso em: 28 abril. 2023.

JavaScript - Overview. Disponível em: <https://www.tutorialspoint.com/javascript/javascript_overview.htm#:~:text=What%20is%20JavaScript%20%3F,language%20with%20object%2Doriented%20capabilities.>. Acesso em: 28 abril. 2023.

LARMAN, Craig; VODDE, Bas Design & Architecture. **Practices for Scaling Lean and Agile Development**. Addison-Wesley Professional, 2010.

KATIE TERRELL HANNA. **Eclipse (Eclipse Foundation)**. Disponível em: <https://www.techtarget.com/searchapparchitecture/definition/Eclipse-Eclipse-Foundation#:~:text=Eclipse%20started%20in%202001%20when,complementing%20Apache's%20open%20source%20community.>. Acesso em: 3 nov. 2023.

MARQUES, Simone. **Arquitetura de Software: por que é tão importante? - Blog UDS**, 2021. Disponível em: <https://uds.com.br/blog/arquitetura-de-software-o-que-e/>. Acesso em: 7 set. 2023.

MICROSOFT**. Estilo de arquitetura de microsserviço - Azure Architecture Center**. Disponível em: <https://learn.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>. Acesso em: 16 set. 2023.

MICROSOFT. **Como funciona o Azure? - Cloud Adoption Framework**. Disponível em: <https://learn.microsoft.com/pt-br/azure/cloud-adoption-framework/get-started/what-is-azure>. Acesso em: 6 nov. 2023.

MELO, D. **O que é HTML? [Guia para iniciantes] – Tecnoblog**. Disponível em: <https://tecnoblog.net/responde/o-que-e-html-guia-para-iniciantes/>. Acesso em: 3 nov. 2023.

MENDES, Antônio. **Arquitetura de Software**: Desenvolvimento orientado para

arquitetura. Rio de Janeiro: Campus Ltda, 2002. ISBN 853521013X

MILINKOVICH, M. **About the Eclipse Foundation | The Eclipse Foundation**. Disponível em: <https://www.eclipse.org/org/>. Acesso em: 28 abril. 2023.

H. P. Nii, **Blackboard Systems: Parts 1 and 2 – AI Magazine**. Vol. 7, No. 3 (pp. 38-53) and Vol. 7, No. 4 (pp. 62-69)

PostgreSQL: About. Disponível em: <https://www.postgresql.org/about/>. Acesso em: 28 abril. 2023.

REDHAT. **Microsserviços**. Disponível em: <https://www.redhat.com/pt-br/topics/microservices>. Acesso em: 12 nov. 2023.

VALENTE, Marco Tulio. **Cap. 7: Arquitetura.** Engenharia de Software Moderna, [*s. d*.]. Disponível em: <https://engsoftmoderna.info/cap7.html>. Acesso em: 10 set. 2023.

VALENTE, Marco Tulio. **O que é uma Arquitetura Hexagonal?**. Engenharia de Software Moderna, [*s. d*.]. Disponível em: https://engsoftmoderna.info/artigos/arquitetura-hexagonal.html#:~:text=Os%20objetivos%20de%20uma%20Arquitetura,mais%20f%C3%A1ceis%20de%20serem%20testados. Acesso em: 13 set. 2023.

SAKOVICH, Natallia**. Microservices vs. Monolithic Architecture Comparison | SaM Solutions**. Disponível em: https://www.sam-solutions.com/blog/microservices-vs-monolithic-real-business-examples/. Acesso em: 12 set. 2023.

SANTOS, L. **Microserviços: dos grandes monólitos às pequenas rotas**. Disponível em: https://medium.com/trainingcenter/microservi%C3%A7os-dos-grandes-mon%C3%B3litos-%C3%A0s-pequenas-rotas-adb70303b6a3. Acesso em: 13 set. 2023.

SHAW, Mary. **Larger scale systems require higher-level abstractions**. Proc. of the International Workshop on Software Specification and Design, Pittsburgh, PA, May

1989, pp. 143-146.

SOUZA, Isaac Felisberto de. **Camadas - Guia dev**. Disponível em: <https://guia.dev/pt/pillars/software-architecture/layers.html>. Acesso em: 31 out. 2023.

O que é o Spring Boot? Disponível em: <https://www.treinaweb.com.br/blog/o-que-e-o-spring-boot>. Acesso em: 3 nov. 2023.

STAFF. **Microsserviços x arquitetura monolítica: entenda a diferença - Viceri**. Disponível em: <https://viceri.com.br/insights/microsservicos-x-arquitetura-monolitica-entenda-a-diferenca/>. Acesso em: 13 set. 2023.

TANEMBAUM, A. S. **Computer Networks**. Prentice Hall, 1988.

TEMPORIN, Tiago. **O que é arquitetura hexagonal.** Aprenda Golang, 2023. Disponível em: <https://aprendagolang.com.br/2023/07/06/o-que-e-arquitetura-hexagonal/>. Acesso em: 13 set. 2023.

TRINDADE, L. **Desvendando a Arquitetura Hexagonal - Tableless - Medium**. Disponível em: <https://medium.com/tableless/desvendando-a-arquitetura-hexagonal-52c56f8824c>. Acesso em: 13 set. 2023.

VS Code - O que é e por que você deve usar? Disponível em: <https://www.treinaweb.com.br/blog/vs-code-o-que-e-e-por-que-voce-deve-usar>. Acesso em: 28 abril. 2023.

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR
Av. Universitária, 1069 • Setor Universitário
Caixa Postal 86 • CEP 74605-010
Goiânia • Goiás • Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br • reitoria@pucgoias.edu.br

RESOLUÇÃO nº 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante LIANDRA DE OLIVEIRA LOPES do Curso de Ciência da Computação, matrícula 2019.1.0028.0095-0, telefone: (62) 99345-4396 e-mail liandragata10@live.com, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do Autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado **Arquitetura de Software**, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto(PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, 14 de dezembro de 2023.

Assinatura do autor: Liandra de O. Lopes

Nome completo do autor: LIANDRA DE OLIVEIRA LOPES

Assinatura do professor–orientador: Eugênio Júlio

Nome completo do professor-orientador: Eugênio Júlio Messala C. Carvalho