LUCAS GABRIEL SILVA

TÉCNICAS E FERRAMENTAS PARA CONSTRUÇÃO DE API'S UTILIZANDO ARQUITETURA SERVERLESS

LAVRAS – MG

2023

LUCAS GABRIEL SILVA

TÉCNICAS E FERRAMENTAS PARA CONSTRUÇÃO DE API'S UTILIZANDO ARQUITETURA SERVERLESS

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do Curso de Sistemas de Informação, para a obtenção do título de Bacharel.

Prof. Dr. Heitor Augustus Xavier Costa
Orientador

LAVRAS – MG

2023

Dedico à toda minha família, principalmente minha mãe Helenice Maria Silva e meu pai Márcio da Silva. Dedico também à minha irmã Gabriela Fernanda Silva, à minha namorada Stéfany Alves dos Santos Neves e todos os meus amigos.

AGRADECIMENTOS

A princípio, agradeço à minha mãe e ao meu pai por todo apoio e todo esforço para oferecer uma vida de qualidade para mim e minha irmã.

Agradeço também a todos os meus familiares, por todo o apoio e desejos de sucesso.

Agradeço à minha namorada pelo apoio e compreensão.

Agradeço a todos os meus amigos, presentes em bons e maus momentos, além de me instigarem sempre nos objetivos profissionais.

Agradeço ao Prof. Dr. Heitor Augustus Xavier Costa por seus ensinamentos diante do curso e por aceitar me orientar e me ajudar no processo de produção deste documento.

E agradeço à Universidade Federal de Lavras, e a todos os seus funcionários, e principalmente todos os professores que tive a oportunidade de ter sido aluno.

RESUMO

Este trabalho apresenta a criação de um software utilizando a arquitetura *serverless*, *Domain-Driven Design* (DDD) e *Clean Architecture*. A arquitetura *serverless* permite que a infraestrutura seja gerenciada automaticamente, aumentando a escalabilidade do sistema de software. Além disso, o DDD permitiu uma modelagem clara do sistema, fornecendo maior visibilidade às regras de negócio do sistema de software. A *Clean Architecture* garantiu uma organização mais eficiente do código. Para provisionar a infraestrutura do sistema de software criado, foi utilizado as ferramentas *Terraform* e *Serverless Framework* para provisionar de forma automática os recursos na *cloud* da AWS. Como resultado, o sistema de software criado apresenta uma escalabilidade eficiente e um código bem organizado, o que contribui para sua manutenção e evolução no futuro.

Palavras-chave: Arquitetura serverless. Domain-Driven Design. Clean Architecture. Terraform. Serverless Framework. AWS.

ABSTRACT

This paper presents the creation of a software using serverless architecture, Domain-Driven Design (DDD) and Clean Architecture. The serverless architecture allows the infrastructure to be automatically managed, increasing the scalability of the software system. In addition, DDD allowed for clear modeling of the system, providing greater visibility into the software system's business rules. The Clean Architecture ensured a more efficient code organization. To provision the infrastructure of the software system created, Terraform and Serverless Framework tools were used to automatically provision resources in the AWS cloud. As a result, the created software system presents an efficient scalability and a well-organized code, which contributes to its maintenance and evolution in the future.

Keywords: Serverless architecture. Domain-Driven Design. Clean Architecture. Terraform. Serverless Framework. AWS.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização e Motivação

A escalabilidade é um desafio crítico na indústria de desenvolvimento de sistemas de software, especialmente com o aumento da demanda dos usuários e o crescimento dos negócios. Como resultado, a necessidade de escalabilidade tem sido um dos principais motivadores para a adoção de arquitetura *serverless* em sistemas de software. De acordo com Castro et al. (2019), a arquitetura *serverless* oferece escalabilidade automática, permitindo que os provedores de serviços escalem automaticamente a capacidade de processamento de acordo com a necessidade da aplicação, sem a necessidade de intervenção humana. Isso pode significar economia de custos, pois só é pago o que se usa e não precisa se preocupar com a capacidade excedente durante períodos de baixa demanda.

Além disso, não basta utilizar uma arquitetura robusta em sistemas de software. É necessário adotar estratégias eficientes para organizá-los ao nível de código, de forma a facilitar o desenvolvimento e a manutenção de funções. Um exemplo de estratégia nesse contexto é utilizar *Clean Architecture*. Segundo Martin (2019) *Clean Architecture* é uma metodologia de projeto de software que busca separar a lógica de negócios da lógica de acesso a dados e da apresentação. Isso permite que o sistema de software seja mais fácil de entender, modificar e testar, além de torná-lo escalável e independente de tecnologias específicas. Com essa abordagem, as camadas dos sistemas de software são organizadas de forma hierárquica, de acordo com sua importância para a lógica de negócios, sendo a camada mais interna, mais abstrata e menos dependente das outras camadas.

1.2 Objetivo

O objetivo deste trabalho foi adquirir conhecimento na utilização de *Clean Architecture* e AWS[1] (*Amazon Web Services*) utilizando as ferramentas *Terraform*[2] e *Serverless Framework*[3]. *Clean Architecture* foi utilizada para organizar os módulos desse sistema de maneira clara e desacoplada, tornando o código mais fácil de manter e evoluir. AWS foi utilizada para gerenciar as configurações necessárias no provedor. Essa escolha deve-se ao fato de AWS ser um provedor

[1] Disponível em: <https://aws.amazon.com/pt/>. Acesso em. 13.fev 2023
[2] Disponível em: <https://www.terraform.io/>. Acesso em. 13.fev 2023
[3] Disponível em: <https://www.serverless.com/>. Acesso em. 13.fev 2023

de nuvem com ampla gama de serviços, incluindo suporte a arquitetura *serverless* e ferramentas de IaC (*Infrastructure as Code*).

Assim, foi desenvolvido um sistema de software, denominado "**BaseLog**", que simula uma API (*Application Programming Interface*) de e-commerce, utilizando *Clean Architecture* e AWS (*Amazon Web Services*).

2 REFERENCIAL TEÓRICO

Neste Capítulo, são apresentados os conceitos, tecnologias e ferramentas que foram usados para auxiliar no desenvolvimento do sistema de software proposto.

2.1 Sistemas Monolíticos e Microsserviços

Os sistemas monolíticos e os microsserviços são modelos de arquitetura de software utilizados para organizar e estruturar um sistema de software. Ambos têm vantagens e desvantagens e são adequados para diferentes tipos de projetos e situações.

2.1.1 Sistemas Monolíticos

De acordo com Newman (2020), um sistema de software é considerado monolítico quando todas as suas funções forem implementadas em conjunto. Esse tipo de sistema torna atividades comuns, como monitoramento e testes fim a fim, mais fáceis de serem realizadas, pois a execução de determinadas funções ocorre no mesmo ambiente. Outro benefício é a reutilização de código, visto que todo o código-fonte se encontra no mesmo local.

Porém, um sistema de software monolítico também proporciona alguns desafios. À medida em que ele "cresce" (torna-se complexo), aumenta as chances de ocorrer "confusão" a respeito em qual lugar determinada função deve ser implementada e quem é responsável por essa parte deste sistema de software. Para Newman (2020), esse problema é conhecido como "desavença nas entregas".

Os sistemas de software monolíticos são geralmente desenvolvidos como um único código-fonte, compilado e executado como um único conjunto de instruções. Como todas as funções são integradas em um único código, existem vantagens e desvantagens com essa abordagem:

- **Vantagens**:
  - ✓ **Maior desempenho**. Pode diminuir a latência de processamento;
  - ✓ **Maior facilidade de implementação**. Permite a implantação do sistema de software de maneira rápida e a reutilização de componentes.
- **Desvantagens**:

- ✓ **Dificuldade de escalabilidade**. Pode dificultar a adição de funções ou a expansão do sistema de software para atender a novas necessidades;
- ✓ **Dificuldade de manutenção**. Pode dificultar a identificação e a correção de erros específicos;
- ✓ **Maior complexidade**. Pode dificultar o entendimento e a manutenção do sistema de software por parte de novos desenvolvedores.

2.1.2 Microsserviços

Segundo Newman (2020), microsserviços são serviços que podem ser implantados de forma independente e são modelados em torno de um domínio de negócio. Dessa forma, cada serviço é responsável por uma tarefa específica e pode ser desenvolvido, implantado e escalonado de forma independente. Existem vantagens e desvantagens com essa abordagem:

- **Vantagens**:
  - ✓ **Flexibilidade**. Os serviços podem ser desenvolvidos e implantados independentemente, permitindo mais flexibilidade e agilidade no desenvolvimento de aplicações;
  - ✓ **Escalabilidade**. Cada serviço pode ser escalonado de forma independente, permitindo otimizar o uso de recursos e garantir a disponibilidade da aplicação;
  - ✓ **Manutenção**. A separação em serviços menores torna mais fácil manter e atualizar o sistema de software, pois é possível fazer alterações em um único serviço sem afetar o funcionamento dos demais.
- **Desvantagens**:
  - ✓ **Complexidade**. A implementação de uma arquitetura de microsserviços pode ser mais complexa do que uma arquitetura monolítica, pois envolve a criação e o gerenciamento de múltiplos serviços;
  - ✓ **Desempenho**. A comunicação entre os serviços pode afetar o desempenho da aplicação, especialmente em situações em que é necessário realizar muitas chamadas de serviço;
  - ✓ **Debug**. Pode ser mais difícil identificar e corrigir problemas em uma aplicação baseada em microsserviços, pois é preciso rastrear as chamadas de serviço entre os diferentes componentes da aplicação.

2.2 Arquitetura *Serverless*

Segundo Castro et al. (2019), arquitetura *serverless* é um serviço fornecido por determinadas *cloud platforms* que oculta o uso do servidor dos desenvolvedores e executa o código do desenvolvedor sob demanda, dimensionando e cobrando automaticamente apenas durante o tempo em que o código está em execução.

Assim, os serviços são executados em uma nuvem e são escalonados automaticamente de acordo com a demanda. Isso significa que as empresas só pagam pelo uso eficiente dos recursos de computação, ao invés de pagar pelo uso de servidores dedicados o tempo todo. A tradução literal de *serverless* é "sem servidor", porém isso não é verdade. É importante destacar que esse tipo de arquitetura ainda é baseada em servidores. Entretanto, seu provisionamento e gerenciamento não são responsabilidade do desenvolvedor, mas do provedor desse serviço. Dessa forma, as tarefas de provisionamento, manutenção e escala da infraestrutura ficam a cargo do provedor.

Para aplicar a arquitetura *serverless*, as empresas podem usar uma plataforma de nuvem que oferece esses serviços, como o AWS ou o *Google Cloud Platform*[1] (GCP). As empresas podem dividir seus aplicativos em pequenas partes chamadas "funções" e executá-las quando são solicitadas. De acordo com Paul Castro (2019), as principais vantagens e desvantagens da arquitetura *serverless* incluem:

- **Vantagens**:
  - ✓ **Baixo custo**. Como só paga pelo uso eficiente dos recursos de computação, a arquitetura *serverless* pode ser mais barata do que pagar por servidores dedicados o tempo todo;
  - ✓ **Escalabilidade**. A nuvem pode escalar automaticamente as funções de acordo com a demanda, o que significa que o aplicativo pode lidar com "picos de tráfego" sem problemas;
  - ✓ **Agilidade**. A arquitetura *serverless* permite que as empresas implementem e gerenciem facilmente seus aplicativos, o que pode acelerar o tempo de lançamento.
- **Desvantagens**:

[1] Disponível em: <https://cloud.google.com/?hl=pt-br>. Acesso em. 13.fev 2023

- ✓ **Complexidade**. Dividir um sistema de software em funções pode ser complexo e pode exigir mudança significativa na forma como esse sistema é projetado e gerenciado;
- ✓ **Dependência da nuvem**. Ao depender de uma nuvem para executar seus sistemas de software, as empresas ficam dependentes dos serviços da nuvem e podem ser afetadas por problemas de disponibilidade ou interrupções;
- ✓ **Custos ocultos**. Embora possa ser mais barata em alguns casos, a arquitetura *serverless* pode levar a custos ocultos se as funções forem executadas frequentemente ou por períodos de tempo mais longos do que o previsto.

2.3 *Domain-Driven Design*

De acordo com Evans (2017), *Domain-Driven Design* é uma maneira de pensar e um conjunto de prioridades, voltado para a aceleração de projetos de software que têm que trabalhar com domínios complicados. Esse conceito foi desenvolvido a partir da experiência do autor como arquiteto de sistemas de software em vários projetos em diferentes contextos. A ideia de *Domain-Driven Design* (DDD) é a modelagem do domínio de negócio ser uma parte crucial do processo de design de software e deve ser tratada com o mesmo cuidado e atenção dedicados às questões técnicas.

Para utilizar o DDD, é importante entender o domínio de negócio da aplicação em questão. Isso inclui compreender os processos, as regras e as relações presentes no domínio, bem como os termos e os conceitos utilizados pelos especialistas do domínio. Além disso, é preciso modelar o domínio de negócio para refletir de maneira precisa e completa os processos, as regras e as relações do domínio. Isso pode ser feito utilizando, por exemplo, Diagramas de Classe e Diagramas de Sequência da UML (*Unified Modeling Language*).

Uma vez que o domínio de negócio foi modelado, o modelo é implementado utilizando uma linguagem de programação orientada a objetos, como Java ou C#, seguindo padrões de projeto (*Design Pattern*). Além disso, é importante garantir que o código implementado seja coerente com o modelo do domínio de negócio e reflita de maneira precisa os processos, as regras e as relações do domínio. O DDD é uma abordagem poderosa para o design de software que pode ajudar a garantir que a aplicação seja coerente com o domínio de negócio e fácil de manter e evoluir. No entanto, é importante lembrar que o DDD é apenas uma das muitas

abordagens possíveis para o design de software e que pode não ser adequado para todos os projetos.

2.4 *Clean Architecture*

Segundo Martin (2019), *Clean Architecture* é um conjunto de princípios de *design de software*, descrito pela primeira vez no artigo intitulado "*The Clean Architecture*". Em 2017, o autor lançou um livro denominado "*Clean Architecture: A Craftsman's Guide to Software Structure and Design*", no qual abordou com mais detalhes o assunto.

A motivação da *Clean Architecture* é ajudar os desenvolvedores a criar sistemas de software fáceis de manter, expandir e testar. *Clean Architecture* baseia-se em princípios que visam separar as preocupações do sistema do software para permitir que mudanças em uma parte desse sistema não afetem outras partes. Isso é conseguido com criação de camadas de abstração responsáveis por um conjunto específico de tarefas. De acordo com Martin (2019), *Clean Architecture* é baseada em quatro camadas principais (Figura 2.1): i) Regras de Negócio da Empresa; ii) Regras de Negócio da Aplicação; iii) Adaptadores de Interface; e iv) Frameworks & Drivers.

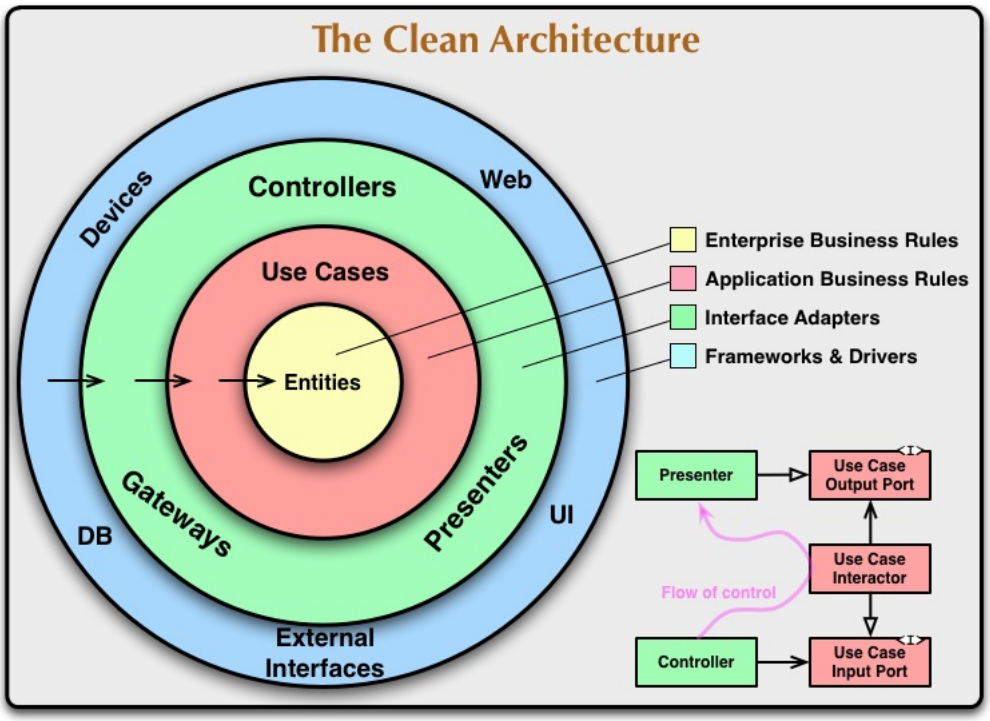
Cada camada é responsável por um conjunto específico de tarefas e comunica-se com as outras camadas por meio de interfaces bem definidas. Além disso, o autor introduz a regra da dependência, afirmando que "As dependências de código-fonte devem apontar apenas para dentro, na direção das políticas de nível mais alto". Isso permite que as camadas sejam substituídas ou alteradas sem afetar as outras camadas do sistema de software.

Vale a pena ressaltar que *Clean Architecture* também enfatiza a importância de escrever código de qualidade e de realizar testes de unidade para garantir a integridade do sistema de software. Além disso, promove a separação de responsabilidades e a criação de sistemas de software modularizados que podem ser facilmente mantidos e modificados à medida que as necessidades do sistema mudam. Segundo Martin (2019), as vantagens de utilizar *Clean Architecture* incluem:

- **Maior flexibilidade**. Ao separar a lógica de negócio da implementação tecnológica, é possível modificar uma sem afetar a outra;
- **Facilidade de manutenção**. Ao organizar o sistema de software em camadas independentes, é mais fácil localizar e corrigir problemas no código;

- **Escalabilidade**. *Clean Architecture* facilita a adição de novas funções ou a integração com outros sistemas de software;
- **Testabilidade**. Ao organizar o sistema de software em camadas, é mais fácil testar cada uma delas individualmente.

Figura 2.1 – *Clean Architecture*



Fonte: Martin (2019)

2.5 *Infrastructure as Code*

O provisionamento de infraestrutura de forma manual é um processo caro e demorado. Isso traz diversas desvantagens, como desperdício de tempo por parte dos desenvolvedores e chances de erros serem cometidos durante o provisionamento. Soluções como virtualização, *containers* e *cloud computing* visam mitigar essas desvantagens. A partir dessas soluções, surgiu o conceito de IaC. Segundo Morris (2016), *Infrastructure as Code* (IaC) é uma prática que trata os componentes de infraestrutura como software, permitindo-os serem versionados, testados e automatizados. Essa abordagem aumenta a confiabilidade, escalabilidade e flexibilidade da infraestrutura.

IaC ajuda as organizações a gerenciar as necessidades de infraestrutura para suas aplicações, melhorando a consistência e reduzindo erros e a necessidade de configuração manual. Atualmente, sem utilizar IaC na implementação, é mais difícil gerenciar os recursos de forma consistente e escalável. Ao automatizar o processo de provisionamento da infraestrutura utilizando IaC, os desenvolvedores não precisam se preocupar em realizar operações manuais para provisionar e gerenciar servidores. Dessa forma, as organizações reduzem custos por economizar o tempo dos desenvolvedores e ganham aumento significativo na velocidade das implementações dos recursos. Além disso, pode-se mitigar os erros de configurações inválidas durante o provisionamento.

2.6 *Terraform*

Segundo a documentação oficial, *Terraform* é uma ferramenta utilizada para provisionar e gerenciar a infraestrutura ao longo de seu ciclo de vida nos principais provedores em nuvem do mercado. É possível gerenciar componentes de baixo nível, como computação, armazenamento e recursos de rede, e componentes de alto nível, como entradas DNS (*Domain Name System*) e recursos SaaS (*Software as a Service*). A criação e o gerenciamento dos recursos pelo *Terraform* são feitos usando APIs fornecidas pelos provedores de nuvem.

O fluxo de trabalho do *Terraform* consiste basicamente em três etapas. A primeira etapa é a gravação, na qual são definidos os recursos a serem utilizados. A segunda etapa consiste no planejamento, na qual *Terraform* cria um plano de execução, descrevendo a infraestrutura a ser criada, atualizada ou destruída com base na infraestrutura existente e em sua configuração. Na terceira etapa, acontece a aplicação da infraestrutura planejada na etapa anterior. Portanto, *Terraform* executa as operações propostas na ordem correta, respeitando quaisquer dependências de recursos.

2.7 *Serverless Framework*

Segundo a documentação oficial, *Serverless Framework* é uma ferramenta utilizada para construir aplicações *serverless*, sendo possível criar e gerenciar funções para executar códigos e responder aos eventos desejados. Dessa forma, os desenvolvedores não precisam se preocupar em provisionar os recursos de forma manual, ganhando agilidade e focando no desenvolvimento de sistemas de software.

Esse *framework* dá suporte a diversas linguagens, como Python[2] e Java[3], e provê suporte aos principais provedores em nuvem do mercado, como Azure[4] e AWS. Além disso, há serviços de monitoramento avançado em sua versão Pro.

2.8 AWS

De acordo com o site oficial, AWS é um provedor de nuvem que oferece diversos serviços e conta com *datacenters* espalhados ao redor do mundo. *Startups*, grandes empresas e órgãos governamentais utilizam os serviços prestados pela AWS. Os serviços fornecidos são desde tecnologias de infraestrutura, como computação, armazenamento e bancos de dados, às tecnologias emergentes, como *machine learning*, inteligência artificial, *data lakes*, análises e Internet das Coisas.

2.8.1 *AWS Lambda*

De acordo com a documentação oficial, *AWS Lambda*[5] é um serviço que permite executar código sem provisionar e gerenciar servidores; basta realizar o *upload* do código e *AWS Lambda* se encarrega dos itens necessários para executar o código, provendo alta disponibilidade e escalabilidade. Esse serviço provê escalabilidade contínua. *AWS Lambda* dimensiona sistemas de software automaticamente executando código em resposta a cada acionamento. O código é executado em paralelo e processa cada acionamento separadamente, escalando de acordo com o tamanho da carga de trabalho. Pode-se otimizar o tempo de execução do código escolhendo o tamanho de memória ideal para cada função *Lambda*.

Existe um tempo necessário para executar a primeira função *Lambda*. Esse conceito é conhecido como inicialização a frio. Porém, pode-se habilitar a simultaneidade provisionada para manter suas funções inicializadas e prontas para responder em questão de poucos milissegundos. O modelo de cobrança é baseado a cada 100ms de execução de código e pela quantidade de vezes que o código é acionado. Dessa forma, o *AWS Lambda* cobra apenas pelo tempo de computação consumido.

[2] Disponível em: <https://www.python.org/>. Acesso em. 13.fev 2023
[3] Disponível em: <https://www.java.com/pt-BR/>. Acesso em. 13.fev 2023
[4] Disponível em: <https://azure.microsoft.com/pt-br/>. Acesso em. 13.fev 2023
[5] Disponível em: <https://docs.aws.amazon.com/lambda>. Acesso em. 13.fev 2023

2.8.2 *AWS DynamoDB*

De acordo com a documentação oficial, *AWS DynamoDB*[6] é um banco de dados NoSQL (não relacional) que oferece alto desempenho para sistemas de software em qualquer escala. É confiável e totalmente gerenciado com segurança, *backups* e restauração integrados, além de armazenamento em cache. Por ser *serverless*, é capaz de expandir e reduzir tabelas automaticamente para ajustar de acordo com a capacidade e manter o desempenho. A disponibilidade e a tolerância a falhas são incorporadas, eliminando a necessidade de projetar esses recursos em sistemas de software.

Em termos de processamento, consegue lidar com mais de 10 trilhões de solicitações por dia e comportar picos de mais de 20 milhões de solicitações por segundo. Além disso, garante tempo de resposta consistente abaixo de 10 milissegundos. As tabelas globais do *DynamoDB* replicam seus dados em várias regiões da AWS para oferecer acesso rápido aos dados. Por fim, é possível configurar o modo de capacidade das tabelas (provisionada e sob demanda) para otimizar custos especificando a capacidade por carga de trabalho ou pagando somente pelos recursos que consumir.

2.8.3 *AWS API Gateway*

De acordo com a documentação oficial, *AWS API Gateway*[7] é uma plataforma de gerenciamento de API que permite criar, publicar e gerenciar APIs. É uma solução robusta e escalável que ajuda a simplificar o processo de integração de *back-end* e *front-end*. Com *AWS API Gateway*, os desenvolvedores podem criar APIs personalizadas, definir rotas, definir autenticação, definir autorização, gerenciar a disponibilidade e monitorar o uso das APIs. Além disso, fornece recursos de gerenciamento de tráfego, incluindo balanceamento de carga, limitação de taxa e gerenciamento de cache.

Outras vantagens incluem a capacidade de integrar facilmente com outros serviços da AWS e criar e publicar APIs em múltiplas regiões. *AWS API Gateway* oferece suporte a vários protocolos, incluindo REST (*Representational State Transfer*), *WebSockets* e HTTP (*Hypertext Transfer Protocol*), o que permite aos desenvolvedores escolher a melhor opção para seus requisitos de aplicativos.

[6] Disponível em: <https://docs.aws.amazon.com/dynamodb/>. Acesso em. 13.fev 2023
[7] Disponível em: <https://docs.aws.amazon.com/apigateway/>. Acesso em. 13.fev 2023

2.8.4 *AWS Systems Manager*

De acordo com a documentação oficial, *AWS Systems Manager*[8] disponibiliza um local central para visualizar e gerenciar seus recursos da AWS para ter visibilidade e controle total sobre suas operações. Pode-se criar grupos lógicos de recursos como aplicativos, separar ambiente de produção e desenvolvimento e criar e gerenciar variáveis de ambiente. Dessa forma, tem-se visão centralizada das configurações e do estado de seus sistemas e permite automação de tarefas comuns, como instalação de sistemas de software, atualizações de segurança e coleta de dados de inventário.

2.9 *Javascript*

Segundo Silva (2015), *JavaScript*[9] é uma linguagem de programação utilizada principalmente para a construção de páginas para a Internet. Foi criada por Brendan Eich, a pedido da empresa Netscape, em meados de 1995. No início, seu principal objetivo era apenas validar formulários HTML (*HyperText Markup Language*). Inicialmente, *JavaScript* foi nomeado como *LiveScript*[10], entretanto Netscape não ficou sozinha durante o desenvolvimento de *JavaScript*. A empresa SUN Microsystems mostrou interesse e forneceu suporte no desenvolvimento da linguagem, acreditando estar construindo grande inovação. Por causa do sucesso da *LiveScript*, a mudança para o nome *JavaScript* foi inevitável, por causa da influência da SUN, que mantinha a linguagem Java. É importante ressaltar que as linguagens *Java* e o *Javascript* são bastante diferentes.

De acordo com Peyrott (2017), Microsoft[11], percebendo o sucesso e o potencial do *Javascript*, resolveu iniciar o desenvolvimento de *JScript*, que executava apenas no Internet Explorer. Isso gerava diversos problemas de incompatibilidade entre os navegadores que, consequentemente, contribuíram para o projeto não ser um sucesso. Percebendo esse problema, ECMA resolveu padronizar a linguagem *JavaScript* para funcionar em todos os navegadores. A partir dessa iniciativa, surgiu o padrão denominado ECMA, sendo batizada de *ECMAScript*. Os sistemas de software desenvolvidos em *JavaScript* são códigos feitos em *ECMAScript*, porém o nome permaneceu *JavaScript* por motivos de marketing, visto que o nome estava consolidado entre os desenvolvedores.

[8] Disponível em: <https://docs.aws.amazon.com/systems-manager/>. Acesso em. 13.fev 2023
[9] Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>. Acesso em. 13.fev 2023
[10] Disponível em: <https://livescript.net/>. Acesso em. 13.fev 2023
[11] Disponível em: <https://www.microsoft.com/pt-br>. Acesso em. 13.fev 2023

2.10 *TypeScript*

Segundo Bright (2012), *TypeScript*[12] começou a ser desenvolvido pela Microsoft em 2012 com o objetivo de adicionar recursos e ferramentas não presentes de forma nativa na linguagem *Javascript*, como tipagem estática. Por isso, não é usualmente considerado como uma nova linguagem de programação, mas um superconjunto de *JavaScript*, pois o código é transpilado[13] (*transpiling*) para *JavaScript* antes de ser executado.

Códigos escritos em *JavaScript* são considerados códigos *TypeScript* válidos por causa do processo de transpilação que ocorre antes de serem executados. Um arquivo *TypeScript* utiliza a extensão ".ts". A principal vantagem de *TypeScript* em relação a *JavaScript* é a adição de recursos importantes e úteis, como tipagem estática e a possibilidade de localizar e corrigir erros em tempo real durante o desenvolvimento. *TypeScript* é um projeto de código aberto que conta com forte participação da comunidade de programadores.

2.11 Node.js

De acordo com a documentação oficial, *Node.js*[14] é um ambiente de execução que permite executar *JavaScript* do lado do servidor. Foi construído a partir da máquina virtual do V8, a mesma utilizada para executar o *JavaScript* no Chrome[15]. Vale ressaltar que *Node.js* é orientado a eventos, seguindo um modelo de entrada e saída não bloqueante. Isso significa que ele é assíncrono e não bloqueia para uma requisição, passando imediatamente para o próximo evento. Esse fato torna o *Node.js* rápido e eficiente, utilizando apenas um *thread*. Além disso, quando iniciado, ele inicia todas as variáveis e funções e aguarda a ocorrência de um evento.

Além disso, *Node.js* conta com diversas bibliotecas e módulos prontos, disponibilizados na Internet, que oferecem diversas funções, facilitando o desenvolvimento de sistemas de software. Para utilizar essas bibliotecas e módulos, é necessário utilizar um gerenciador de pacotes, por exemplo, *Node Package Manager*[16] (NPM).

[12] Disponível em: <https://www.typescriptlang.org/>. Acesso em. 13.fev 2023
[13] Converter código fonte em outro por um processo de compilação, ou seja, o programador escreve código fonte em uma linguagem que, após ser compilada, gerará código equivalente em outra linguagem.
[14] Disponível em: <https://nodejs.org/en/>. Acesso em. 13.fev 2023
[15] Disponível em: <https://www.google.com/intl/pt-BR/chrome/>. Acesso em. 13.fev 2023
[16] Disponível em: <https://www.npmjs.com/>. Acesso em. 13.fev 2023

2.12 Testes de Unidade

De acordo com Santos (2019), testes de unidade são uma forma de garantir a qualidade do código, verificando o comportamento de pequenas partes isoladas do sistema de software, como funções ou métodos específicos, garantindo que determinada unidade de código esteja funcionando conforme o esperado.

Os testes de unidade ajudam a identificar problemas de forma precoce, antes que eles se propaguem para outras partes do sistema de software, o que facilita a correção e reduz o custo do processo de desenvolvimento. Esse tipo de teste pode ser utilizado para verificar a validação de dados, comportamento esperado em diferentes cenários, entre outros aspectos importantes para a qualidade do sistema de software. Por isso, é importante investir tempo e recursos na criação e execução de testes de unidade, buscando sempre identificar e corrigir problemas antes que eles possam causar prejuízos maiores.

3 MÉTODO DE PESQUISA

O objetivo deste Capítulo é apresentar como os conceitos, tecnologias e ferramentas foram utilizados no desenvolvimento do sistema de software proposto.

3.1 Tipo de Pesquisa

O tipo de pesquisa pode ser definido como pesquisa aplicada (tecnológica). Segundo Lakatos e Marconi (2021), uma pesquisa aplicada tem como característica fundamental o interesse na aplicação, utilização e consequências práticas dos conhecimentos. Dessa forma, seu objetivo é alcançar a inovação em um processo ou produto com base em uma necessidade ou demanda percebida.

3.2 Procedimentos

O primeiro passo foi realizar revisões literárias a respeito dos assuntos necessários para a modelagem e a implementação do **BaseLog**. Além disso, foram consultados alguns sistemas de software que utilizam a mesma tecnologia e possuem o código fonte aberto. Após a aquisição do conhecimento, foram definidos os seguintes componentes para a implementação do **BaseLog**: i) **Linguagem de programação**: para a implementação do sistema de software; ii) ***Cloud***: para disponibilizar o sistema de software de forma online; iii) **Banco de dados**: para armazenar os dados do sistema de software; e iv) **IaC**: para automatizar o provisionamento das infraestruturas necessárias.

3.3 Componentes Utilizados

Para selecionar os componentes de um sistema de software, é importante avaliar vários fatores, incluindo performance, escalabilidade e custo. Alguns aspectos a serem considerados ao escolher os componentes incluem:

- **Performance**. Os componentes devem ser capazes de atender às demandas de performance do sistema de software. Isso pode incluir a velocidade de processamento, a capacidade de armazenamento e a largura de banda de rede necessária;
- **Escalabilidade**. O sistema de software deve ser capaz de lidar com o aumento da carga de trabalho e da quantidade de usuários sem perder desempenho. Isso pode ser alcançado

escolhendo componentes que possam ser facilmente adicionados ou removidos conforme necessário;

- **Custo**. É importante avaliar o custo total de propriedade dos componentes, incluindo o custo inicial, o custo de manutenção e o custo de atualização. Alguns componentes podem ter custo inicial mais alto, mas podem ter custo de manutenção mais baixo a longo prazo.

Além disso, é importante considerar outros fatores, como a compatibilidade com outros componentes do sistema de software, a facilidade de uso e o suporte técnico disponível. Os critérios descritos anteriormente foram essenciais na escolha dos seguintes componentes utilizados na implementação do **BaseLog**: i) **Linguagem de programação**: *Javascript*, utilizando o superset *Typescript* para facilitar o desenvolvimento; ii) ***Cloud***: AWS; iii) **Banco de dados**: *AWS DynamoDB*; e iv) **IaC**: *Serverless Framework* e *Terraform*.

3.4 Escolha da Arquitetura

De acordo com Newman (2020), existem algumas considerações a serem feitas ao escolher se um sistema de software deve ser projetado como um sistema monolítico ou como microsserviços, por exemplo:

- **Escala**. Microsserviços são mais fáceis de escalar do que sistemas de software monolíticos, pois cada serviço pode ser escalado individualmente de acordo com as necessidades;

- **Flexibilidade**. Microsserviços permitem mais flexibilidade no desenvolvimento, pois os serviços podem ser desenvolvidos, testados e implantados independentemente uns dos outros;

- **Manutenção**. Microsserviços permitem mais facilidade na manutenção e atualização de sistemas de software, pois os serviços podem ser atualizados individualmente sem afetar o sistema de software inteiro;

- **Complexidade**. Microsserviços podem ser mais complexos de serem projetados e gerenciados do que sistemas de software monolíticos, pois envolvem maior quantidade de componentes interconectados;

- **Custo**. O desenvolvimento de microsserviços pode ser mais custoso do que o de sistemas de software monolíticos, pois envolve a criação de mais componentes e a necessidade de gerenciá-los de forma independente.

Em resumo, microsserviços são uma boa opção para sistemas de software que precisam de escalabilidade, flexibilidade e facilidade de manutenção, mas podem ser mais complexos e custosos de serem projetados e gerenciados. Sistemas de software monolíticos são mais simples de serem projetados e gerenciados, mas podem ser menos escaláveis e flexíveis. Dessa forma, considerando as vantagens e as desvantagens em relação a esses dois tipos de arquitetura, optou-se para a construção do **BaseLog** com a arquitetura de sistemas de software monolíticos. Isso se deve ao fato do BaseLog ter como foco a aplicação dos conceitos de modelagem DDD e a organização em *Clean Architecture*.

Para a modelagem do **BaseLog**, foram utilizados os conceitos propostos pelo DDD. De acordo com Evans (2017), para implementar o DDD é importante:

- **Compreender o domínio do sistema**. Esta é a etapa mais importante. É necessário compreender profundamente as necessidades, as regras e o fluxo de negócios do domínio;
- **Modelar a linguagem Ubíqua**. Utilizar uma linguagem unificada e compartilhada para modelar o domínio, que ajude a comunicar as ideias com clareza;
- **Definir entidades, agregados e contextos**. Tem-se insumos para a construção do diagrama arquitetural do sistema de software.

Vale ressaltar que DDD é complexo e não necessariamente é preciso aplicar todas as estratégias disponibilizadas, pois o importante é trazer clareza em relação às regras de negócio de um sistema de software. Assim, o uso de DDD tende a ser mais benéfico para sistemas de software que possuem muitas funções. Visando à aplicação do DDD no **BaseLog**, foi realizado um levantamento das necessidades: i) Cadastro de usuário; ii) Login de usuário; iii) Gerenciamento de produtos; iv) Vendas de produtos; e v) Notificação para administradores. Para evitar dúvidas em relação aos termos utilizados na definição dos contextos, foi utilizada a estratégia de estruturar a linguagem Ubíqua para **BaseLog**. Dessa forma, os seguintes termos foram definidos:

- **Conta**. Representa as informações relativas a um usuário do **BaseLog**;
- **Autenticação**. Processo de validar o acesso de usuários ao **BaseLog**;

- **Autorização**. Processo de validar se usuários têm permissão para acessar determinada função do **BaseLog**;
- **Produto**. Representa determinado item disponibilizado para a venda no **BaseLog**;
- **Catálogo**. Representa os produtos disponibilizados no **BaseLog**;
- **Checkout**. Processo de compra de produtos no qual ocorre o processamento do pagamento;
- **Transação**. Contém informações de pagamentos processados pelo **BaseLog**;
- **Notificação**. Representa uma mensagem enviada pelo **BaseLog** para determinado administrador.

Com base nesses termos, foram definidos os seguintes contextos representados por módulos no **BaseLog**:

- **Módulo *Authentication***. Responsável pelos processos de cadastro de usuários, login e autorização;
- **Módulo *Catalog***. Responsável pelo controle do domínio dos produtos;
- **Módulo *Checkout***. Responsável por iniciar o processo de compra de um produto, bem como a atualização do *status* do pedido;
- **Módulo *Payment***. Responsável por processar as transações financeiras e pela comunicação com o *gateway* de pagamento;
- **Módulo *Notification***. Responsável por enviar notificações via *e-mail* ao administrador do sistema de software.

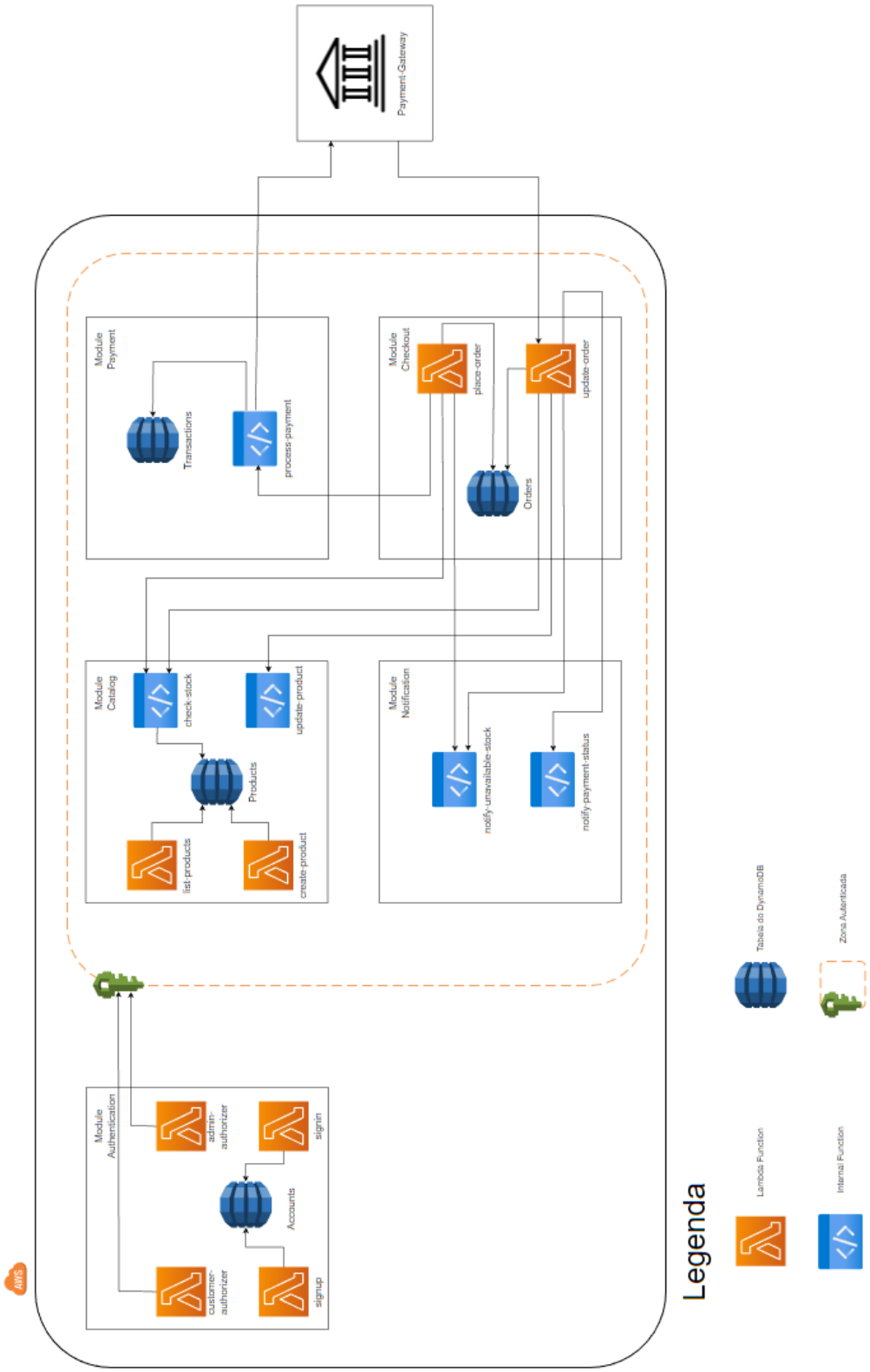
Para implementar esses módulos, foram definidas as entidades:

- ***Accounts***. Responsável por lidar com as regras de negócio relativas à autenticação e autorização;
- ***Products***. Responsável por lidar com as regras de negócio relativas aos produtos;
- ***Orders***. Responsável por lidar com as regras de negócio relativas aos pedidos;

- ***Transactions***. Responsável por lidar com as regras de negócio relativas às transações financeiras.

Após identificar os módulos e as entidades, foi construído um diagrama apresentado na Figura 3.1. O objetivo foi representar a comunicação e organização dos módulos, apresentando as funções e tabelas do banco de dados utilizados.

Figura 3.1 – *Diagrama do* ***BaseLog***



Fonte: Autor

3.5 Utilizando *Clean Architecture*

Para visar *Clean Architecture* no desenvolvimento do **BaseLog**, foram adotadas as seguintes camadas: i) Camada de domínio; ii) Camada de aplicação; iii) Camada de apresentação; e iv) Camada de *framework*. Vale ressaltar que não existe padronização para o nome dessas camadas, ficando a cargo das pessoas responsáveis pela modelagem do sistema de software adotarem e seguirem um padrão.

3.5.1 Camada de Domínio

A camada de domínio é a camada central da *Clean Architecture* e é responsável por conter as regras de negócio da aplicação. Ela é completamente independente das camadas externas. A comunicação entre a camada de domínio e as camadas externas é feita por meio de contratos (interfaces), que definem os métodos que podem ser chamados e os dados que podem ser acessados. Isso permite que a camada de domínio seja facilmente testada e reutilizada em outras aplicações. No desenvolvimento do **BaseLog**, foram implementados os seguintes componentes:

- ***Entities***. Contém as entidades que representam os elementos do **BaseLog**, como produtos, clientes, pedidos, etc. Esses elementos são responsáveis por armazenar e validar os dados, bem como realizar as operações de negócio;

- ***Errors***. Contém os erros personalizados do **BaseLog** utilizados nas entidades para facilitar o lançamento de exceção em caso de erro ao longo da execução;

- ***Utils***. Contém elementos que facilitam a implementação dos demais componentes dessa camada, como adaptadores para bibliotecas externas;

- ***Value Objects***. Contém os tipos personalizados utilizados para melhor representação das propriedades das entidades, evitando utilizar os tipos primitivos. Ao aplicar o *pattern Value Object*, pode-se garantir que os valores atribuídos a determinada entidade são consistentes pois são validados ao serem instanciados. Dessa forma, é garantido que as regras de negócio estabelecidas para a entidade são seguidas e o seu estado é consistente ao longo das operações. Além disso, pode-se reaproveitar esses *Value Objects*, evitando a duplicação de código ao construir as validações para as entidades.

3.5.2 Camada de Aplicação

A camada de aplicação é responsável por intermediar a comunicação entre camada de domínio (lógica de negócio) e a camada de apresentação (adaptadores de interface). Ela recebe as requisições do usuário e as encaminha para a camada de negócios, que realiza o processamento e retorna o resultado para a camada de aplicação. Além disso, essa camada pode conter lógica de validação de dados e autenticação de usuários, garantindo a segurança da aplicação. Além disso, é responsável por tratar os erros e exceções gerados pela camada de domínio e pelas requisições externas a aplicação, como outros sistemas de software. Dessa forma, garante-se a integridade e o funcionamento correto do sistema de software. No desenvolvimento do **BaseLog**, foram implementados os seguintes componentes:

- ***Dtos***. Contém as interfaces que definem as entradas e as saídas para cada caso de uso implementado nessa camada;
- ***Errors***. Contém os erros personalizados que representam os possíveis erros que podem ocorrer ao acessar recursos externos ao **BaseLog**;
- ***Repositories***. Contém as interfaces que representam os repositórios utilizados para a persistência de dados no **BaseLog**;
- ***Usecases***. Contém a implementação dos casos de uso, responsáveis por orquestrar a comunicação entre a camada de domínio, a camada de apresentação e recursos externos;
- ***Services***. Contém as interfaces que representam os *services* responsáveis pela lógica de consumo de funções externas ao **BaseLog**;
- ***Utils***. Contém elementos que facilitam a implementação dos demais componentes dessa camada, como adaptadores para bibliotecas externas.

3.5.3 Camada de Apresentação

A camada de apresentação é responsável por receber as solicitações do usuário ou de outros sistemas e encaminhá-las para a camada de aplicação. Por sua vez, a camada de aplicação processa a solicitação e retorna a resposta para a camada de apresentação. Essa camada deve ser independente da camada de aplicação e dos outros módulos do sistema de software, o que

significa que ela não deve conter lógica de negócio ou acesso a dados. Sua função restringe-se a mapear os dados a serem recebidos e enviados. No desenvolvimento do **BaseLog**, foram implementados os seguintes:

- ***Controllers***. Contém os controladores responsáveis por receber a requisição, validar o formato dos dados e mapear o retorno recebido pela camada de aplicação;
- ***Validations***. Contém os validadores responsáveis por verificar se os dados estão de acordo com o contrato estabelecido pela camada de aplicação. É importante ressaltar que esse tipo de validação não inclui lógica de negócio; pois contraria a função atribuída à camada de apresentação. Nesse cenário, as validações basicamente resumem-se a verificar a tipagem dos campos, visto que a linguagem escolhida para a construção do **BaseLog** não é fortemente tipada.

3.5.4 Camada de *Framework*

A camada de *framework* é a camada mais externa do sistema de software e é responsável por gerenciar as suas interações com o mundo exterior. Isso inclui a interface com o usuário, a comunicação com outros sistemas e a integração com outras bibliotecas e serviços. Em *Clean Architecture*, a camada de *framework* deve ser dependente da camada de aplicação e da camada de apresentação, além de ser totalmente substituível. Na prática, essa camada fornece as instâncias dos componentes a serem utilizados pelo sistema de software. Além disso, a camada de *framework* deve ser a mais simples possível, concentrando-se apenas nas tarefas de interface e integração, enquanto a lógica de negócios é delegada para a camada de aplicação. Isso ajuda a manter a arquitetura limpa e facilita a manutenção do sistema de software. No desenvolvimento do **BaseLog**, foram implementados os seguintes componentes:

- ***Database***. Contém as implementações dos repositórios definidos na camada de aplicação;
- ***Facade***. Contém a implementação do *design pattern Facade* utilizado para a comunicação entre os módulos do **BaseLog**;
- ***Factories***. Contém as funções responsáveis por criar as instâncias dos elementos a serem utilizados pelo **BaseLog**;
- ***Functions***. Contém as funções responsáveis por realizar o mapeamento dos eventos recebidos por *lambda function* e que devem ser enviados para a camada de apresentação;

- ***Services***. Contém as implementações dos *services* definidos na camada de aplicação.

3.6 Utilizando *Infrastructure as Code*

A infraestrutura para o desenvolvimento do **BaseLog** na AWS foi planejada e implementada utilizando duas ferramentas: *Terraform* e *Serverless Framework*. Juntas, elas permitem a construção e o gerenciamento de sistemas de software altamente escaláveis e confiáveis na AWS. Terraform foi utilizado para provisionar os seguintes recursos: i) Tabelas do *AWS DynamoDB*; ii) Políticas de acesso do *AWS IAM*; e iii) Parâmetros do *AWS Systems Manager*. *Serverless Framework* foi utilizado para provisionar os seguintes recursos: i) *Lambdas Functions*; e ii) *API Gateway*. A divisão dos arquivos de *Terraform* e do *Serverless Framework* foi realizada de acordo com a estrutura de módulos definida na arquitetura do **BaseLog**. Pode-se fazer *deploys* em duas áreas distintas: desenvolvimento e produção. Essa separação é crucial, pois permite testar novas funções e correções antes de disponibilizá-las para os usuários.

3.7 Utilizando Testes de Unidade

Visando à aplicação de testes de unidade no **BaseLog**, foi utilizada a biblioteca Jest[1]. De acordo com a documentação oficial, Jest é um *framework* de testes em *JavaScript* com foco na simplicidade, sendo um dos mais populares atualmente. Os testes desenvolvidos foram divididos por módulos, replicando a estrutura de módulos definidas para o **BaseLog**, facilitando a organização do código. Além disso, foram definidas algumas classes auxiliares que fornecem instâncias dos objetos utilizados no texto, evitando a duplicação de código.

Figura 3.2 – Exemplo de testes utilizando a biblioteca Jest

```
import { PriceInCents } from '@catalog/domain/value-objects';
import { InvalidPriceInCentsError } from '@catalog/domain/errors';

describe('PriceInCents Value Object', () => {
  test('should create a new PriceInCents', () => {
    const value = 1000;
    const priceInCents = PriceInCents.create(value);
    expect(priceInCents).toBeDefined();
    const getValue = priceInCents.getValue();
    expect(getValue).toBe(value);
  });

  test('should throw if send a invalid value', () => {
    const value = -100;
    expect(() => PriceInCents.create(value)).toThrow(InvalidPriceInCentsError);
    expect(() => PriceInCents.create(value)).toThrow(`The price "${value}" is invalid`);
  });
});

```

Fonte: Autor

[1] Disponível em: <https://jestjs.io/pt-BR/>. Acesso em. 13.fev 2023

A figura 3.2 contém dois testes feitos utilizando a biblioteca Jest. O primeiro teste verifica se é possível instanciar a classe *PriceInCents* e acessar seus atributos ao fornecer um número inteiro e positivo para o método *create*, que é responsável por retornar uma instância da classe. Já o segundo teste verifica se ocorre uma exceção ao fornecer um número inteiro e negativo para o método *create*.

4 RESULTADOS

Neste Capítulo, são apresentados os resultados obtidos com o desenvolvimento do sistema de software proposto.

4.1 Funções

Para garantir que as funções planejadas sejam implementadas de forma adequada, as funções do **BaseLog** foram disponibilizadas via *endpoints* HTTP. Isso significa que todas as requisições para o **BaseLog** podem ser feitas por meio de uma interface web. Além disso, permite aos desenvolvedores trabalharem de forma mais eficiente e colaborativa, pois as requisições e as respostas são claramente definidas e padronizadas. As funções do **BaseLog** são descritas nas próximas seções.

4.1.1 *Sign Up*

Essa função é responsável por criar uma conta de usuário não administrador. É necessário fornecer as seguintes informações:

- ***userName***. Representa o nome do usuário. É validado se não existe a presença de números no valor fornecido;
- ***email***. Representa o *e-mail* utilizado pelo usuário na autenticação. É validado se o valor fornecido possui o formato de *e-mail* válido e se não existe outro usuário utilizando este *e-mail*;
- ***password***. Representa a senha utilizada pelo usuário na autenticação. É validada se o valor fornecido satisfaz os seguintes critérios: i) Possui pelo menos uma letra maiúscula; ii) Possui pelo menos uma letra minúsculo; iii) Possui pelo menos um número; iv) Possui pelo menos um caracter especial; e v) Possui um comprimento mínimo de 8 caracteres.

4.1.2 *Sign In*

Essa função é responsável por validar as credenciais de acesso fornecidas pelo usuário. O BaseLog verifica se existe uma combinação de *e-mail* e senha com os mesmos valores fornecidos e, em caso positivo, retorna um *token* de acesso. Vale destacar que, por motivos de

segurança, não é especificado qual campo está incorreto em caso de credenciais inválidas. É necessário fornecer as seguintes informações:

- ***email***. Representa o *e-mail* utilizado pelo usuário na autenticação;
- ***password***. Representa a senha utilizada pelo usuário na autenticação.

4.1.3 *Create Product*

Essa função é responsável por cadastrar os produtos disponibilizados para vendas. É necessário fornecer as seguintes informações:

- ***productName***. Representa o nome do produto. É validado se não existe a presença de números no valor fornecido;
- ***priceInCents***. Representa o valor do produto em centavos. É validado se o valor fornecido é um número inteiro positivo;
- ***amount***. Representa a quantidade disponível do produto. É validado se o valor fornecido é um número inteiro positivo;

4.1.4 *List Products*

Essa função é responsável por listar os produtos disponibilizados para vendas. Não é necessário fornecer propriedades para essa função.

4.1.5 *Place Order*

Essa função é responsável por processar uma venda, gerando uma ordem. É necessário fornecer as seguintes informações:

- ***productId***. Representa o identificador único do produto;
- ***amount***. Representa a quantidade de produtos vendidos. É validado se o valor fornecido é um número inteiro positivo;

Após a validação das informações fornecidas, a função verifica se a quantidade de produtos determinada está disponível no estoque. Em caso positivo, é enviada uma solicitação para o módulo *Payment* gerar uma transação, que, em seguida, encaminha a solicitação para o *gateway* de pagamento. Caso o produto não esteja com estoque disponível, é enviada uma notificação para o administrador do sistema relatando o ocorrido.

4.1.6 *Update Order*

Essa função é responsável por atualizar o *status* das vendas. É necessário fornecer as seguintes informações:

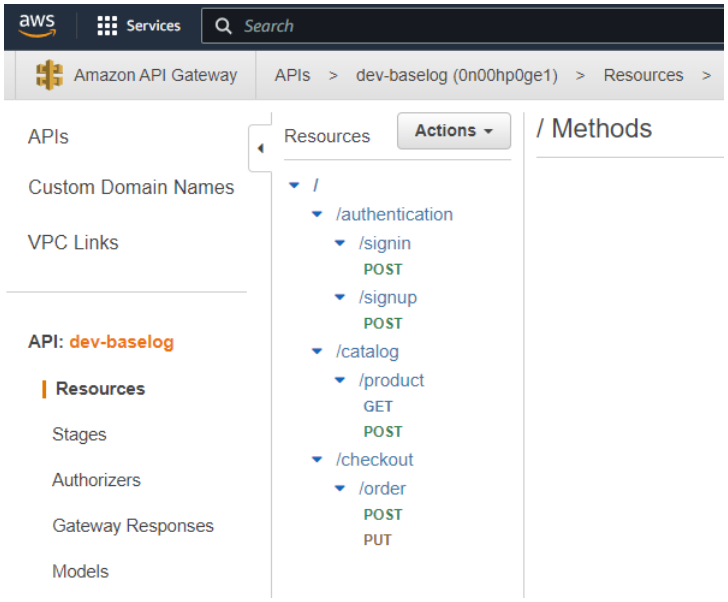
- ***orderId***. Representa o identificador único da ordem;
- ***orderStatus***. Representa o *status* da ordem. Os valores possíveis são: i) PENDING; ii) APPROVED; iii) RECUSED; iv) CANCELED; e v) FAILURED;
- ***transactionStatus***. Representa o *status* da transação. Os valores possíveis são: i) PENDING; ii) APPROVED; iii) RECUSED; iv) CANCELED; v) PENDING_CHARGEBACK; e vi) CHARGED_BACK.

4.2 Hospedagem

Visando à disponibilização do **BaseLog** na web, foram utilizadas as ferramentas *Terraform* e *Serverless Framework* para provisionar a infraestrutura necessária na AWS. Dessa forma, o **BaseLog** foi disponibilizado em três passos:

- **1° passo**. Executar o comando *terraform apply*. Esse passo consiste em provisionar, via *Terraform*, as tabelas do *DynamoDB*, SNS utilizado para enviar as notificações e as variáveis de ambiente;
- **2° passo**. Executar o comando *sls deploy*. Esse passo consiste em provisionar, via *Serverless Framework*, as funções *lambdas* e *API Gateway*, disponibilizando os *endpoints*. A figura 4.1 apresenta o *dashboard* do *Api Gateway* na AWS após a execução do comando citado;
- **3° passo**. Alterar o valor da variável de ambiente utilizada para gerar o *token* de acesso da aplicação. Por motivos de segurança, esse valor não é adicionado diretamente no código, pois trata-se de uma informação crítica de segurança da aplicação.

Figura 4.1 – *Dashboard* do *Api Gateway* na AWS

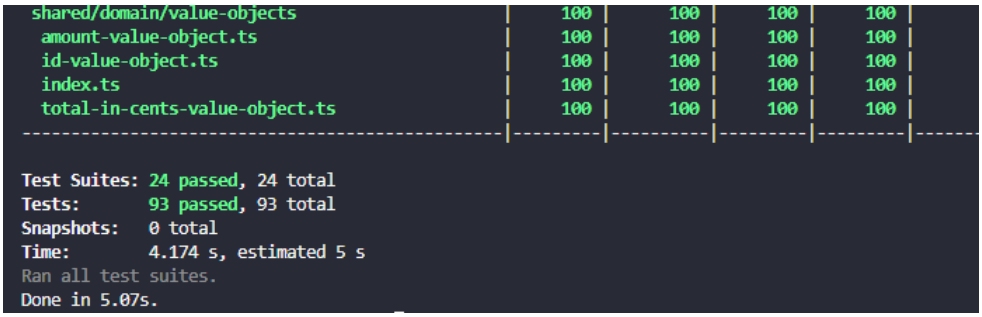


Fonte: Autor

4.3 Testes de Unidade

Para garantir a qualidade do código do **BaseLog**, foram implementados testes de unidade que verificam se as funções atendem aos requisitos estabelecidos. A figura 4.2 apresenta o resultado dos 93 testes de unidade desenvolvidos, atingindo uma cobertura total das funções testadas. É importante ressaltar que o tempo médio de execução dos testes é de 5 segundos, o que é considerado um tempo razoável para a quantidade de testes realizados.

Figura 4.2 – Resultado dos testes de unidade desenvolvidos



Fonte: Autor

5 CONSIDERAÇÕES FINAIS

Em primeiro lugar, é importante destacar a importância da arquitetura *serverless* para o desenvolvimento de sistemas de software escaláveis. A capacidade de distribuir automaticamente recursos de acordo com a demanda sem a necessidade de preocupar-se com a infraestrutura subjacente permite que as equipes de desenvolvimento se concentrem no que realmente importa: o desenvolvimento de recursos e funções.

Em segundo lugar, a modelagem com DDD e a organização do código com *Clean Architecture* são fundamentais para a construção de sistemas de software de alta qualidade. A utilização de DDD permite a criação de modelos de domínio claros e coerentes, facilitando a compreensão do negócio e a implementação de recursos. A estruturação do código com *Clean Architecture* proporciona organização bem definida, facilitando a manutenção e o desenvolvimento de novas funções.

Por fim, pode-se concluir que a combinação da arquitetura *serverless*, da modelagem com DDD e da organização do código com *Clean Architecture* é uma opção viável e eficiente para o desenvolvimento de sistemas de software escaláveis, confiáveis e de alta qualidade. Essa combinação permite o desenvolvimento de sistemas de software que atendem às necessidades do negócio, atendento a satisfação dos usuários e a eficiência da equipe de desenvolvimento.

5.1 Trabalhos futuros

Este trabalho foi realizado com o objetivo de explorar os conceitos de arquitetura *serverless*, DDD e *Clean Architecture*. Embora tenha sido possível identificar vantagens no desenvolvimento de sistemas de software, é importante destacar que o **BaseLog** não possui muitas funções. Portanto, uma sugestão de continuidade deste trabalho é a implementação de novas funções e recursos no **BaseLog** para avaliar a eficiência e escalabilidade da arquitetura *serverless*. Nesse contexto, também seria interessante avaliar a capacidade de escala do **BaseLog**. Para isso, poderia ser realizado testes de carga a fim de obter métricas relativas a escalabilidade do sistema de software.

Além disso, pode-se realizar comparações entre a arquitetura *serverless* e outras abordagens de arquitetura de software. Essa comparação pode avaliar as vantagens e desvantagens de cada abordagem, identificando qual é a mais adequada para cada situação.

Outra sugestão de continuidade é a integração da arquitetura *serverless* com outras tecnologias e ferramentas, como inteligência artificial, *machine learning* e *big data*. A combinação

dessas tecnologias/ferramentas pode resultar em soluções avançadas e eficientes, com a capacidade de processar grandes volumes de dados e oferecer recursos avançados aos usuários.

Em resumo, o trabalho realizado apresenta diversas possibilidades de continuidade, permitindo aprofundar a compreensão sobre os conceitos de arquitetura *serverless*, DDD e *Clean Architecture* e aplicá-los de forma eficiente na construção de sistemas de software escaláveis e confiáveis.

REFERÊNCIAS

BRIGHT, P. **Microsoft TypeScript: the JavaScript we need, or a solution looking for a problem?** 2012. Disponível em: <https://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/>. Acesso em: 20 dez. 2022.

CASTRO, P. et al. **The server is dead, long live the server: Rise of Serverless Computing, Overview of Current State and Future Trends in Research and Industry**. 2019. Disponível em: <https://arxiv.org/abs/1906.02888/>. Acesso em: 03 fev. 2023.

EVANS, E. **Domain-Driven Design: Atacando as complexidades no coração do software - 3ª edição**. [S.l.]: Alta Books, 2017. ISBN 9788550800653.

LAKATOS, E. M.; MARCONI, M. de A. **Metodologia do Trabalho Científico 9ª edição**. [S.l.]: Atlas, 2021. ISBN 9788597026535.

MARTIN, R. C. **Arquitetura Limpa: O guia do artesão para estrutura e design de software**. [S.l.]: Alta Books, 2019. ISBN 9788550804606.

MORRIS, K. **Infrastructure as Code: Managing Servers in the Cloud**. [S.l.]: O'Reilly Media, 2016. ISBN 9781491924358.

NEWMAN, S. **Migrando sistemas monolíticos para microsserviços: Padrões evolutivos para transformar seu sistema monolítico**. [S.l.]: Novatec, 2020. ISBN 9786586057041.

PEYROTT, S. **A Brief History of JavaScript**. 2017. Disponível em: <https://auth0.com/blog/a-brief-history-of-javascript/>. Acesso em: 20 dez. 2022.

SANTOS, F. **Testes unitários: uma abordagem para garantir a qualidade do código**. 2019. Disponível em: <https://blogdaprogramacao.com/testes-unitarios>. Acesso em: 02 fev. 2023.

SILVA, G. **O que é e como funciona a linguagem JavaScript?** 2015. Disponível em: <https://canaltech.com.br/internet/O-que-e-e-como-funciona-a-linguagem-JavaScript/>. Acesso em: 20 dez. 2022.