UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ALENCAR DA COSTA

Notifications as a Service: API para envio de notificações através de múltiplos canais e provedores

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof[a]. Dra. Renata de Matos Galante

Porto Alegre
2023

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Profª. Patricia Helena Lucas Pranke
Pró-Reitor de Graduação: Profª. Cíntia Inês Boll
Diretora do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Diretora da Escola de Engenharia: Profª. Carla Schwengber Ten Caten
Coordenador do Curso de Engenharia de Computação: Prof. Cláudio Machado Diniz
Bibliotecário-Chefe do Instituto de Informática: Alexsander Borges Ribeiro
Bibliotecária-Chefe da Escola de Engenharia: Rosane Beatriz Allegretti Borges

Dedico este trabalho ao meu irmão,
Vinícius (in memorian), que sempre esteve
presente em minha memória e em meu coração.

AGRADECIMENTOS

Aos meus pais, Eloí Borba da Costa e Regina Beatriz da Costa, por todos os sacrifícios realizados para que eu e minha irmã tivéssemos acesso às melhores oportunidades de educação e por sempre nos incentivarem e apoiarem a aproveitá-las ao máximo. Agradeço pelo apoio incondicional que possibilitou esta e outras experiências.

À minha irmã, Maria Clara da Costa, por todo apoio e por todos os momentos de descontração, os quais tornaram esta experiência mais alegre.

Aos grandes amigos que a graduação me proporcionou, Filipe Faria Dias, Gabriel Augusto Engel, Matheus Woeffel Camargo e Raphael Scherpinski Brandão, por tornarem a graduação mais divertida. Em especial, ao Matheus, por sempre me ouvir e compartilhar.

À Professora Renata de Matos Galante pela orientação e pelos incentivos, bem como a confiança depositada para que a escolha deste trabalho fosse possível.

A todos os participantes dos testes de usabilidade que compartilharam seu tempo para avaliar o sistema desenvolvido.

Por fim, agradeço a todo o corpo docente e a todos os servidores da UFRGS, os quais contribuíram, de forma direta ou indireta, em minha jornada acadêmica.

RESUMO

Este trabalho apresenta o contexto do mercado de notificações e suas necessidades, tendo como objetivo a implementação das funcionalidades fundamentais para um sistema de notificações agregador de canais e provedores. Para isso, avaliamos as circunstâncias em que notificações são utilizadas, bem como a abordagem aplicada por soluções já existentes, a fim de identificar os principais requisitos exigidos por empresas de *software* e seus usuários quanto ao envio, automação, gerenciamento e recebimento de notificações. Apoiados nisso, desenvolvemos um sistema de notificações com uma API REST única, a qual abstrai a complexidade vinculada ao envio de notificações através de múltiplos canais e provedores a partir da utilização de um sistema de *templating* baseado em blocos de conteúdo. Além disso, implementamos recursos para automação de fluxos personalizados de envio e gerenciamento de preferências de destinatários, assim como a disponibilização de dados sobre o engajamento de destinatários com as notificações. Consequentemente, conseguimos aplicar testes de usabilidade com potenciais usuários e verificar se o sistema de notificações permite uma rápida integração com sua API REST, bem como o fácil envio de notificações para múltiplos canais e provedores. Em decorrência disso e da adoção de boas práticas, os participantes afirmaram a preferência por esta solução e a economia de tempo e esforço que a acompanha. Assim, o sistema de notificação desenvolvido demonstrou-se uma solução efetiva para as necessidades do mercado e com bom potencial de adoção, bem como potencial para ser utilizado como referência para implementações de *softwares* semelhantes.

Palavras-chave: Notificações. e-mail. SMS. API REST.

Notifications as a Service: API for sending notifications across multiple channels and providers

ABSTRACT

This thesis presents the context of the notifications market and its needs, aiming to implement the fundamental functionalities for a notification system that aggregates channels and providers. To this end, we evaluated the circumstances in which notifications are used, in addition to the approach applied by existing solutions, in order to identify the main requirements demanded by software companies and their users regarding the sending, automation, management, and receipt of notifications. Based on this, we developed a notification system with a single REST API, which abstracts the complexity of sending notifications through multiple channels and providers using a templating system based on content blocks. In addition, we implemented features for automating customized delivery flows and for managing recipient preferences, as well as providing data about recipient engagement with notifications. Consequently, we were able to apply usability tests with potential users and verify that the notification system allows fast integration with its REST API, including easy sending of notifications to multiple channels and providers. As a result of this and the adoption of best practices, participants affirmed their preference for this solution and the time and effort savings that come with it. Thus, the notification system developed has proven to be an effective solution for the market needs and with good adoption potential, along with the potential to be used as a reference for similar software implementations.

Keywords: notifications, e-mail, SMS, REST API.

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend as a Service
CLI	Command-line Interface
DX	Developer Experience
FaaS	Function as a Service
FCM	Firebase Cloud Messaging
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
JSON	JavaScript Object Notation
REST	Representational State Transfer
SNS	Simple Notification Service
SQS	Simple Queue Service
URI	Uniform Resource Identifier
YAML	YAML Ain't Markup Language

LISTA DE FIGURAS

Figura 2.1 Representação básica de um sistema de notificações com suporte a múltiplos canais e provedores. 21

Figura 3.1 Amazon API Gateway atuando como *trigger* de uma função Lambda. 33
Figura 3.2 Comparação da performance de consultas entre DynamoDB e bancos relacionais a medida que o volume de dados armazenado cresce. Fonte: (DEBRIE, 2021) 38
Figura 3.3 Fluxo de disponibilização de alterações realizadas no DynamoDB para o DynamoDB Streams. 39
Figura 3.4 Acesso a recursos da aplicação através do Amazon API Gateway e AWS Lambda com a utilização do Amazon Cognito com uma *user pool*. 43

Figura 5.1 Visão geral do sistema de notificações e os serviços que o compõem. 57
Figura 5.2 Arquitetura de alto nível do serviço de *Identity and Access Management* do sistema de notificações. 60
Figura 5.3 Visão geral do serviço de *Identity and Access Management* (IAM) do sistema de notificações. 61
Figura 5.4 Diagrama de Entidade Relacionamento das entidades presentes no serviço de IAM do sistema de notificações. 62
Figura 5.5 Arquitetura do serviço de *Identity and Access Management* (IAM) do sistema de notificações. 63
Figura 5.6 Processo de *sign-up* de novos usuários no serviço de *Identity and Access Management* (IAM) do sistema de notificações. 64
Figura 5.7 Tela de *sign-up* do Cognito. 65
Figura 5.8 Processo de autorização para rotas voltadas para aplicações *front-end*. 66
Figura 5.9 Processo de autorização para rotas voltadas para outros *softwares*. 67
Figura 5.10 Arquitetura de alto nível do serviço de *Notification* do sistema de notificações. 70
Figura 5.11 Visão geral do serviço de *Notification* do sistema de notificações. 71
Figura 5.12 Transições de *status* presentes no ciclo de vida de uma notificação. 80
Figura 5.13 Diagrama de Entidade Relacionamento das entidades presentes no serviço de *Notification* do sistema de notificações. 81
Figura 5.14 Detalhes da arquitetura do serviço de *Notification* do sistema de notificações. 82
Figura 5.15 Representação do padrão *Transactional Outbox* utilizando serviços da AWS conforme implementado no sistema de notificações. 85
Figura 5.16 Detalhes da arquitetura do serviço *Notification* para tarefa de envio de notificações. 87
Figura 5.17 Fluxograma das operações realizadas pela função *workload-maker* do serviço de *Notification*. 88
Figura 5.18 Fluxograma das operações realizadas pela função *workload-distributor* do serviço de *Notification*. 89
Figura 5.19 Fluxograma das operações realizadas pela função *workflow-runner* do serviço de *Notification*. 90
Figura 5.20 Fluxograma das operações realizadas pela função *notification-sender* do serviço de *Notification*. 91
Figura 5.21 Fluxograma das operações realizadas para validar se as preferências do destinatário permitem o envio da notificação. 92

Figura 5.22 Fluxograma das operações realizadas por funções de integração com provedores de envio de notificação. 93
Figura 5.23 Detalhes de como é feito o recebimento de eventos dos provedores. 95
Figura 5.24 Detalhes de como é feita a execução de uma etapa de *delay* de um *Workflow*. 96
Figura 5.25 Arquitetura de alto nível do serviço de *Analytics* do sistema de notificações. 104
Figura 5.26 Visão geral do serviço de *Analytics* do sistema de notificações. 105
Figura 5.27 Diagrama de Entidade Relacionamento das entidades presentes no serviço de *Analytics* do sistema de notificações. 107
Figura 5.28 Detalhes da arquitetura do serviço de *Analytics* do sistema de notificações. 108
Figura 5.29 Tela do aplicativo desenvolvido para testar o envio de *push notifications*. 112

Figura 6.1 Exemplo de requisição para criar *template* com definições para e-mail e *push notifications*. 116
Figura 6.2 Exemplo de requisição para criar novo *workflow*. 117
Figura 6.3 Exemplo de requisição para definir etapas de um *workflow*. 118
Figura 6.4 Exemplo de requisição de envio de notificações usando um *workflow*. 119
Figura 6.5 Exemplo de requisição consultar o histórico de envio filtrando por *workload*. 120
Figura 6.6 E-mail de boas-vindas da empresa enviado ao destinatário. 121
Figura 6.7 Na esquerda, a *push notification* de boas-vindas da empresa e, na direita, o SMS de boas-vindas do time de desenvolvedores. 121

Figura 7.1 Gráfico dos resultados da pesquisa para questão 1. 125
Figura 7.2 Gráfico dos resultados da pesquisa para questão 2. 126
Figura 7.3 Gráfico dos resultados da pesquisa para questão 3. 126
Figura 7.4 Gráfico dos resultados da pesquisa para questão 4. 127
Figura 7.5 Gráfico dos resultados da pesquisa para questão 5. 127
Figura 7.6 Gráfico dos resultados da pesquisa para questão 6. 128
Figura 7.7 Gráfico dos resultados da pesquisa para questão 7. 128
Figura 7.8 Gráfico dos resultados da pesquisa para questão 8. 129
Figura 7.9 Gráfico dos resultados da pesquisa para questão 9. 129
Figura 7.10 Gráfico dos resultados da pesquisa para questão 10. 130
Figura 7.11 Gráfico dos resultados da pesquisa para questão 11. 130
Figura 7.12 Gráfico dos resultados da pesquisa para questão 12. 131
Figura 7.13 Gráfico dos resultados da pesquisa para questão 13. 131
Figura 7.14 Gráfico dos resultados da pesquisa para questão 14. 132
Figura 7.15 Gráfico dos resultados da pesquisa para questão 15. 132
Figura 7.16 Gráfico dos resultados da pesquisa para questão 16. 133
Figura 7.17 Gráfico dos resultados da pesquisa para questão 17. 133
Figura 7.18 Gráfico dos resultados da pesquisa para questão 18. 134
Figura 7.19 Gráfico dos resultados da pesquisa para questão 19. 134
Figura 7.20 Gráfico dos resultados da pesquisa para questão 20. 135
Figura 7.21 Gráfico dos resultados da pesquisa para questão 21. 135

LISTA DE TABELAS

Tabela 4.1 Análise comparativa das funcionalidades e características presentes nas soluções de *Courier*, *Knock* e *SuprSend*. 53

Tabela 5.1 Análise comparativa das funcionalidades e características presentes neste trabalho e nas soluções de *Courier*, *Knock* e *SuprSend*. 113

SUMÁRIO

1 INTRODUÇÃO **14**
2 FUNDAMENTAÇÃO TEÓRICA **16**
2.1 Panorama do mercado de notificações **16**
2.2 Sistema de notificações **17**
2.3 *Representational State Transfer* (REST) **20**
2.3.1 Cliente-servidor 21
2.3.2 *Stateless* 22
2.3.3 Cache 22
2.3.4 Interface uniforme 23
2.3.5 Sistema baseado em camadas 24
2.3.6 Código sob demanda 24
2.3.7 Recursos 24
2.4 *Serverless* **25**
2.5 *Infrastructure as Code* (IaC) **28**
3 TECNOLOGIAS UTILIZADAS **30**
3.1 *Serverless Framework* **30**
3.2 AWS Lambda **32**
3.3 Amazon *Simple Queue Service* (SQS) **34**
3.4 Amazon *Simple Notification Service* (SNS) **35**
3.5 Amazon DynamoDB **36**
3.5.1 DynamoDB Streams 39
3.5.2 Motivação para escolha do DynamoDB 40
3.6 Amazon EventBridge Scheduler **41**
3.7 Amazon API Gateway **41**
3.8 Amazon Cognito **42**
3.9 AWS CloudFormation **44**
3.10 PlanetScale e Vitess **44**
3.10.1 Vitess 44
3.10.2 PlanetScale 46
3.10.3 Motivação para escolha 46
3.11 Node.js **47**
4 TRABALHOS RELACIONADOS **48**
4.1 *Courier* **48**
4.2 *Knock* **49**
4.3 *SuprSend* **51**
4.4 Análise comparativa **52**
5 DESENVOLVIMENTO E IMPLEMENTAÇÃO **55**
5.1 Visão Geral **55**
5.2 Provisionamento e implantação **57**
5.3 Padronizações da API **58**
5.4 Módulo de *Identity and Access Management* (IAM) *Service* **59**
5.4.1 Visão Geral 59
5.4.2 Entidades e Banco de Dados 60
5.4.2.1 *Organization* 61
5.4.2.2 Key 61
5.4.2.3 Representação no Banco de Dados 62
5.4.3 Arquitetura e Funcionamento 62
5.4.3.1 Processo de *sign-up* 64

5.4.3.2 Processo de autenticação e autorização 66
5.4.4 Rotas da API 67
5.4.4.1 Autenticação 67
5.4.4.2 Gerenciamento de chaves de acesso (*API keys*) 67
5.5 Módulo de *Notification Service* 68
5.5.1 Visão Geral 69
5.5.2 Entidades e Banco de Dados 70
5.5.2.1 *Organization* 71
5.5.2.2 *Integration* 71
5.5.2.3 *Element* 72
5.5.2.4 *Template* 73
5.5.2.5 *Recipient* 74
5.5.2.6 *Preference Section* 75
5.5.2.7 *Preference Topic* 75
5.5.2.8 *Workflow* 76
5.5.2.9 *Workload* 77
5.5.2.10 *Workload Run* 78
5.5.2.11 *Notification* 79
5.5.2.12 *Notification Event* 80
5.5.2.13 Representação no Banco de Dados 81
5.5.3 Arquitetura 81
5.5.3.1 *Transactional Outbox* 85
5.5.3.2 Fluxo de envio de notificação 86
5.5.3.3 Eventos dos provedores 94
5.5.3.4 Interrupção e agendamento de *Workload Runs* 94
5.5.4 Rotas da API 97
5.5.4.1 Autenticação e Autorização 97
5.5.4.2 Gerenciamento de integrações 97
5.5.4.3 Gerenciamento de *templates* 98
5.5.4.4 Gerenciamento de *workflows* 99
5.5.4.5 Gerenciamento de preferências 100
5.5.4.6 Gerenciamento de destinatários 101
5.5.4.7 Envio de notificações 103
5.6 Módulo de *Analytics Service* 103
5.6.1 Visão Geral 104
5.6.2 Entidades e Banco de Dados 106
5.6.3 Arquitetura 107
5.6.4 Rotas da API 109
5.6.4.1 Histórico de Notificações 109
5.7 Integrações com provedores 111
5.8 Análise comparativa com os trabalhos relacionados 112
6 DEMONSTRAÇÃO 115
6.1 Construção de um *template* 115
6.2 Construção de um *workflow* 116
6.3 Execução do *workflow* 117
6.4 Visualização do histórico e notificações 117
7 EXPERIMENTOS E RESULTADOS 122
7.1 Protocolo do Experimento 122
7.2 Resultados dos Experimentos 125
7.2.1 Perfil dos usuários 125
7.2.2 Avaliação do sistema 127

7.3 Análise de boas práticas **134**
7.4 Análise Geral dos Resultados.. **138**
8 CONCLUSÃO **140**
REFERÊNCIAS.. **142**
ANEXO A — REQUISIÇÕES DO EXPERIMENTO **146**
ANEXO B — DOCUMENTAÇÕES DA API.. **156**
ANEXO C — FORMULÁRIO DA PESQUISA **157**

1 INTRODUÇÃO

Notificações, sejam elas voltadas para marketing ou não, são exploradas por grande parte dos produtos de software no mercado para informar os usuários de algum evento relevante. Para alcançar os usuários e entregar as notificações, existem diversos canais disponíveis para serem explorados por equipes de software, por exemplo, e-mail, SMS, *push notification*, notificações *in-app* e *chat*. Dado que cada tipo de notificação e canal possuem características e necessidades diferentes, equipes de software utilizam múltiplos canais de notificação para alcançar melhores taxas de resposta do usuário. A taxa de resposta para notificações voltadas para atividades de marketing, segundo *Gartner, Inc.*, é de 45% para SMS e apenas 6% para e-mail (PEMBERTON, 2016). Logo, é evidente a necessidade de que produtos de software ofereçam notificações por meio de diferentes canais conforme as preferências dos usuários e as taxas de resposta para o produto em cada canal e tipo de notificação.

O requisito funcional de enviar notificações através de múltiplos canais é um desafio enfrentado por grande parte das equipes de software e possui o potencial de se tornar ainda maior a medida que novos dispositivos inteligentes são introduzidos no mercado e incluem mecanismos de notificação (HASAN, 2022). Além disso, para cada possibilidade de canal existem diversos provedores disponíveis no mercado, os quais oferecem o serviço de envio de notificações. A vasta gama de opções, em conjunto com requisitos não-funcionais de resiliência e custos, pode tornar necessário a integração com múltiplos provedores para um mesmo canal. Estes requisitos, segundo Goode (2021c), tornam a implementação de um sistema de notificações uma tarefa complexa, bem como a manutenção de um número significativo de integrações com APIs externas. Diante destas complexidades, equipes de software podem demandar excessivas horas de trabalho para implementar um sistema de notificações próprio e que não faz parte do que é entendido como núcleo do produto. O tempo reservado para tarefas como essa é valioso para produtos de software em estágio inicial, como *startups* e, por isso, alternativas ao desenvolvimento de uma solução própria podem ser bastante atrativas.

Goode (2021c) descreve que um sistema de notificações eficiente e performático depende de outros fatores além do suporte a múltiplos canais e provedores. O sistema deve ser resiliente e escalável para suportar altos volumes de notificações e a imprevisibilidade dos picos de envio. Tudo isso sem a ocorrência de perda de notificações e com suporte a tratativas em casos de falhas de envio. Além disso, mecanismos de *templating* e

design de notificações, da mesma forma que relatórios analíticos sobre a entrega de notificações e engajamento dos usuários, são importantes para o gerenciamento do produto. Já para os desenvolvedores, uma API intuitiva que abstrai as complexidades de cada canal e provedor é importante para facilitar a manutenção e aumentar a produtividade. Portanto, para ser eficaz, um sistema de notificações requer funcionalidades complexas com grande capacidade de consumir um tempo valioso de equipes focadas em desenvolver produtos de software.

O software construído por este trabalho trata-se um sistema de notificações que visa facilitar e agilizar a implementação do suporte a notificações em produtos de software via uma API REST. Com isso, deseja-se estudar e compreender os requisitos e funcionalidades vinculadas a um sistema de notificações para que, trabalhos futuros, possam estender suas capacidades e trazer novas características inovadoras ao mercado de envio de notificações.

Este sistema almeja alcançar a escalabilidade e resiliência necessária para suportar altos volumes e picos de notificações por meio de uma arquitetura baseada em eventos e da abordagem de desenvolvimento *serverless*. Para abstrair a complexidade inerente a cada um dos canais e provedores, foi desenvolvida uma API simples e intuitiva, a qual permite que equipes de software se integrem rapidamente e passem a suportar múltiplos canais de notificações imediatamente. Dessa forma, desenvolvedores poderão focar na implementação das regras específicas de seus produtos, ao invés de dedicarem tempo e dinheiro para construir um sistema de notificações próprio.

Assim, este trabalho está dividido em oito Capítulos, incluindo a introdução. No Capítulo seguinte é apresentada a fundamentação teórica a respeito do mercado de notificações, dos principais conceitos atrelados a um sistema de notificações e conceitos importante vinculados às tecnologias utilizadas. No Capítulo 3, são apresentados os trabalhos relacionados e que serviram de inspiração para este trabalho, bem como uma comparação entre suas funcionalidades e recursos. Já na Capítulo 4, são apresentadas as tecnologias utilizadas no desenvolvimento do sistema de notificações e as motivações para suas escolhas. Em seguida, no Capítulo 5, é descrito o desenvolvimento e implementação do sistema, bem como sua arquitetura, módulos e detalhes da API. No Capítulo 6, é realizada uma demonstração das principais funcionalidades do sistema de notificações implementado. Então, no Capítulo 7, são apresentados os experimentos realizados com usuários e seus resultados. Finalmente, no Capítulo 8, são apresentadas as conclusões deste trabalho e apontadas sugestões para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este Capítulo visa apresentar os principais conceitos relacionados ao contexto em que o trabalho está inserido, bem como os principais conceitos relacionados às tecnologias utilizadas. Primeiramente, na Seção 2.1, descreve-se o mercado de notificações, os requisitos e serviços que costumam estar presentes em um sistema de notificações. Para isso, apresenta-se dados atuais da utilização de notificações por produtos de *software* e perspectivas para o futuro do mercado. Em seguida, na Seção 2.2, apresenta-se em detalhes a definição de um sistema de notificações e a descrição de seus requisitos e funcionalidades principais, as quais são exploradas ao longo deste trabalho. Por fim, as Seções 2.3, 2.4 e 2.5 descrevem, respectivamente, conceitos sobre REST, *serverless* e *Infrastructure as Code*, os quais são fundamentais para diversas tecnologias abordadas na implementação do trabalho.

2.1 Panorama do mercado de notificações

Notificações, sejam elas enviadas por e-mail, SMS ou *push notification*, permitem que aplicações avisem os seus usuários de novos eventos na aplicação, mensagens recebidas e lembretes. Ao considerar apenas *push notifications* entregues para usuários de *smartphones*, Visuri et al. (2019) indica que os usuários recebem até 60 notificações por dia. O alto volume de envio de notificações por parte de aplicações também pode ser notado no relatório produzido pela Airship (2021), um provedor do serviço de envio de *push notifications*, o qual demonstra um volume de mais 600 bilhões de notificações enviadas para cerca de 2 bilhões de usuários no ano de 2020. Já os dados de 2020 da *Twilio SendGrid*, apresentados por Moorhead (2022), demonstram que o provedor de e-mail enviou mais de 1 trilhão de notificações através deste canal. Esses dados, embora não deem uma visão global do mercado de notificações, servem como indicativo do alto volume de notificações enviadas por aplicações e processadas por múltiplos provedores.

Além do alto volume de envio de notificações já praticado pelos provedores nos principais canais disponíveis, os resultados apresentados por provedores do mercado demonstram um aumento no uso de notificações por parte das aplicações. O relatório de Zakowicz (2022), com base nos dados 2021 do provedor de notificações *Omnisend*, demonstrou que o número de notificações automatizadas enviadas por aplicações via SMS aumentou em 258% em 2021. Simultaneamente, a introdução de novos dispositivos co-

nectados à Internet ao mercado tem o potencial de amplificar o volume de notificações enviadas por aplicações de *software*. Os dados de Hasan (2022) apontam que, em 2022, é esperado um crescimento de 14% no número de dispositivos *IoT* conectados. Consequentemente, aumenta-se a quantidade de possíveis gatilhos para envio de notificações, bem como o número de possíveis destinatários.

A medida que novos dispositivos são introduzidos no mercado, novos canais de notificações podem surgir, bem como provedores que oferecem o serviço de envio de notificações através do canal. Logo, ao aumentar o volume de notificações e o número de possíveis canais e provedores, também aumenta-se o desafio que equipes de *software* têm para se conectar aos seus usuários por meio de múltiplos canais de notificações em busca dos melhores índices de conversão. Isto é, o esforço para manter a escalabilidade de um sistema de notificações próprio e para dar manutenção a múltiplos canais e provedores pode se tornar inviável. Assim, equipes de desenvolvedores podem utilizar um sistema de notificações externo para removerem a responsabilidade de darem manutenção a um componente do sistema que, comumente, não faz parte do núcleo do *software* sendo desenvolvido.

2.2 Sistema de notificações

Produtos de *software* e equipes de desenvolvimento costumam utilizar notificações para comunicar eventos aos seus usuários e mantê-los engajados ao produto. De acordo com Goode (2021b), esse caso de uso requer alguns requisitos para que o envio de notificações ocorra conforme o esperado e que as notificações enviadas possam servir de dados de entrada para análises da experiência e comportamento dos usuários. Um requisito importante é o suporte a múltiplos canais de distribuição de notificações e provedores destes canais. Isso permite que o produto possa alcançar com maior eficiência seus usuários conforme o contexto, preferências do usuário e tipo de usuário, o que também torna a experiência do usuário com o produto mais agradável.

O caso de uso descrito, segundo Carney (2021), exige que o envio de notificações seja resiliente e escalável para que notificações sejam recebidas pelos usuários sem ocorrência de falhas e grande atrasos. Entretanto, alcançar os níveis de qualidade desejados em um sistema de notificações proprietário com esses requisitos pode ser uma tarefa custosa para equipes de desenvolvimento focadas em soluções específicas não diretamente relacionadas a notificações (GUPTA, 2022; SEELY, 2022). Por conta disso, deseja-se

implementar um sistema de notificações que seja fácil e rápido de se integrar. Assim, equipes de desenvolvimento poderão utilizar uma solução pronta para atender seus requisitos, evitando a reescrita de código e o aumento da superfície de código para ser mantido, bem como aumenta o foco e a velocidade no desenvolvimento do produto.

Goode (2021b, 2021c) descreve que um sistema de notificação com as capacidades mencionadas requer diversos serviços, como mecanismos de *templating*, integração com provedores de notificações para múltiplos canais de distribuição, lógica de roteamento de notificações para decidir o melhor canal e provedor a ser usado no envio, gerenciamento de preferência do usuário e *logging* para análise de utilização e coleta de métricas. Em conjunto, esses serviços permitem que o suporte ao envio de notificações por meio de múltiplos canais e provedores seja implementado com agilidade e facilidade. Além disso, um sistema com tais serviços elimina a complexidade envolvida no processo de criação de cada notificação e torna a experiência dos desenvolvedores ao enviar notificações através da integração com o sistema consistente entre todos os canais e provedores, uma vez que a complexidade de cada um destes é abstraída em uma interface única.

O suporte a múltiplos canais e provedores é importantíssimo nesse contexto, pois permite que usuários sejam alcançados através do canal mais adequado para ele e para o contexto da notificação. Ao compor o sistema de notificações com os serviços citados, conquista-se a vantagem de possuir uma ferramenta única para monitorar e analisar notificações através dos múltiplos canais e provedores por meio da agregação dos dados, tornando a análise e consulta muito mais acessível. Logo, um sistema de notificações que possua tais componentes pode atender aos requisitos desejados por produtos de *software* e desenvolvedores.

Porém, Goode (2021b, 2021c) afirma que o sistema de notificações possuir as funcionalidades necessárias não é suficiente, requer-se que a qualidade dos serviços ofertados seja a esperada e que o sistema tenha o comportamento previsto. Para isso, é fundamental que o sistema de notificações proposto tenha atributos de confiabilidade e escalabilidade suficientemente bons para o serviço continuar sendo ofertado com qualidade mesmo em momentos de pico de utilização. A perda de mensagens, a falta de garantia de entrega da mensagem e a entrega duplicada fazem com que o sistema não seja confiável o suficiente para ser adotado em ambientes com alto volume de notificações e imprevisibilidade de utilização. Logo, prefere-se que em tais situações as notificações sejam entregues com um pequeno atraso, ao invés da perda de notificações. Para isso e para suportar a imprevisibilidade do volume de utilização, o sistema de notificações precisa ser escalável e

isto deve ser transparente para os desenvolvedores que se integram ao sistema. Portanto, a disponibilidade, confiabilidade e escalabilidade são propriedades que devem receber a devida atenção no sistema proposto.

Assim, como a escalabilidade do sistema deve ser transparente para os desenvolvedores que se integram a ele, a complexidade dos canais e provedores oferecidos por ele também devem ser transparentes. Para isso, é responsabilidade do sistema de notificações abstrair tais complexidades e as diferenças entre as APIs dos diferentes provedores para fornecer uma interface única e simples que agregue todos os canais e provedores. Essa é a característica-chave do sistema que fará com que desenvolvedores economizem muito tempo de desenvolvimento de seus produtos de *software*. Uma organização pode desejar utilizar um novo provedor motivada por vantagens comerciais, por exemplo. Essa alteração, através do sistema de notificações proposto, deverá ser simples e não necessitar alterações no código de integração. Logo, é importante que o sistema de notificações tenha suporte a um amplo catálogo de canais e provedores. Para isso, além de possuir uma interface única para o cliente, o sistema de notificações deve ser de fácil extensão, ou seja, deve permitir que os desenvolvedores que mantém o sistema de notificações possam adicionar suporte a novos canais e provedores com facilidade.

Já em relação às funcionalidades secundárias do sistema de notificação, mas que possuem alto valor para o sistema, podemos citar a análise do histórico de notificações enviadas e o gerenciamento de preferência dos usuários. Dado que notificações são usadas para alcançar os usuários finais de um produto, é importante que o sistema de notificações agregue as notificações enviadas através dos canais e provedores em um formato simples que permita o fácil entendimento e análise dos resultados de engajamento. Assim, as organizações podem entender como seus usuários interagem com seus produtos.

Além disso, o sistema deve conter capacidades de gerenciamento de *templates*. Assim, usuários do sistema podem definir conteúdos de notificações que serão reutilizados diversas vezes para engajar e interagir com destinatários. Por meio de funcionalidades como essa, usuários sem conhecimento técnico podem colaborar na construção de modelos de notificações utilizados pela aplicação.

Da mesma forma, o sistema de notificações deve conseguir decidir dinamicamente o melhor canal e provedor para enviar a notificação conforme as regras de negócio aplicadas, contexto da notificação e preferências do usuário por meio de lógicas de roteamento de notificações (GOODE, 2021a). Mesmo que exista a possibilidade de enviar simultaneamente a notificação para múltiplos canais, essa provavelmente não é a melhor estra-

tégia para alcançar o usuário e, inclusive, pode gerar uma experiência desagradável para eles. Pielot, Church and Oliveira (2014) demonstra que além da alta média do número de notificações recebidas por usuários diariamente, grande parte das notificações recebidas não são consideradas importantes pelos usuários e podem desencadear emoções negativas. Heinisch et al. (2022) especificam que as interrupções inapropriadas causadas pelo alto número de notificações podem gerar incômodo e ansiedades nos usuários e, consequentemente, diminuir a produtividade sentida por ele. Isso demonstra a importância de permitir que os próprios usuários determinem o interesse ou não em determinadas notificações, bem como os canais preferidos para serem comunicados, como uma forma de melhorar a experiência do usuário.

Ao verificar todas as funcionalidades, requisitos funcionais e não-funcionais descritos, identifica-se que a construção de um sistema que atenda a todas as demandas das empresas de *software*, dos destinatários e das instituições reguladores do mercado não é uma tarefa simples. Devido aos grandes esforços necessários para enviar notificações, e estão vinculados a custos de tempo, desenvolvedores e financeiros, empresas de *software* podem dar preferências a soluções prontas ofertadas em um modelo de serviço, ao invés de construir sua própria solução. Esta decisão pode significar grandes economias para empresas que não desejam arcar com todos os custos atrelados à complexidade de desenvolver e manter um sistema de envio de notificações, em especial, *startups*, as quais poderão focar exclusivamente no desenvolvimento de inovações de seus produtos.

Com base nos principais requisitos descritos anteriormente, de maneira simplificada, pode-se representar um sistema de notificações como um componente que se integra a múltiplos provedores de envio de notificações e abstrai toda complexidade de envio, infraestrutura e gerenciamento de destinatários em uma API simples. Através desta API, os desenvolvedores e outros sistemas podem realizar o envio de notificações de forma transparente, já que toda complexidade é gerenciada pelo próprio sistema. Essa representação simplificada de um sistema de notificações pode ser visualizada na Figura 2.1.

2.3 *Representational State Transfer* (REST)

Representational State Transfer (REST) é um conjunto de restrições arquiteturais desenvolvidas por Roy Fielding. As restrições impostas servem como guia para o desenvolvimento de aplicações que necessitem se comunicar em redes complexas como a Internet. Através da adoção do guia e, consequentemente, das restrições, as aplicações se

Figura 2.1: Representação básica de um sistema de notificações com suporte a múltiplos canais e provedores.

beneficiam do alto grau de escalabilidade, separação entre cliente e servidor, bem como independência de tecnologias. A seguir, as restrições presentes em uma arquitetura REST são descritas em detalhes conforme as definições do autor (AMAZON, 2022; FIELDING, 2000).

2.3.1 Cliente-servidor

A adoção de um modelo cliente-servidor como uma das restrições da definição de REST foi motivada pelo princípio atrelado a este modelo arquitetural: *separation of concerns*. Através da separação das responsabilidades da interface de usuário e do armazenamento de informações, obtém-se como benefício uma maior portabilidade e flexibilidade da interface de usuário, a qual pode ser desenvolvida para diferentes plataformas em aplicações separadas, por exemplo, um aplicativo, um site ou outro servidor atuando como cliente. Ao mesmo tempo, ganha-se em escalabilidade do servidor, que passa a ser mais simples e possuir menos componentes e responsabilidades (AMAZON, 2022; FIELDING, 2000). Através do modelo cliente-servidor, o cliente passa a ter acesso aos recursos oferecidos pelo servidor e, no contexto de aplicações REST, a comunicação entre as partes costuma ser realizada através do protocolo HTTP (REDHAT, 2020).

2.3.2 *Stateless*

Essa restrição indica que a comunicação entre cliente e servidor deve ser naturalmente *stateless*, ou seja, o servidor não armazena nenhum estado relacionado às requisições recebidas das aplicações cliente. Isso faz com que toda requisição enviada ao servidor precise conter todas as informações necessárias para o processamento da requisição e que o servidor não utilize nenhum tipo de contexto previamente armazenado durante o processamento de requisições anteriores. Logo, todo e qualquer estado da comunicação entre cliente e servidor deve ser mantido na aplicação cliente. Através dessa limitação, toda requisição é tratada de forma independente e completamente isolada de outras requisições, o que faz com que clientes possam solicitar recursos ao servidor sem precisar seguir nenhuma ordem específica de requisições, pois o servidor sempre tem todos os dados necessários para completar a requisição (AMAZON, 2022; FIELDING, 2000).

Em sua tese, Fielding (2000) argumenta que através da adoção dessa restrição, a visibilidade no monitoramento da aplicação é simplificada e melhorada, dado que ferramentas de observabilidade não precisam de múltiplas requisições para compreender as ações sendo efetuadas, visto que toda requisição contém o contexto necessário. Além disso, indica que a confiabilidade da aplicação é simplificada devido a maior facilidade de recuperação de falhas parciais, já que não há necessidade de manutenção de estado anterior a falha, pois não existe estado. Por fim, o servidor também beneficia-se de maior escalabilidade, visto que a implementação é simplificada ao não precisar armazenar contexto e a possibilidade de liberar recursos logo após a utilização. Por outro lado, alguns *trade-offs* precisam ser considerados. Ao não armazenar o contexto no servidor, o desempenho da rede durante a comunicação pode ser degradada devido ao envio de repetitivo de informações extras de contexto nas requisições. Além disso, ao armazenar o contexto na aplicação cliente, a complexidade de implementação do cliente aumenta.

2.3.3 Cache

A especificação desenvolvida por Fielding (2000) indica que as aplicações REST devem suportar estratégias de cache. Essa característica diz respeito a possibilidade de o cliente ou serviços intermediários armazenarem algumas das respostas do servidor para reduzir o tempo de resposta em requisições futuras. As respostas dadas pelo servidor devem indicar para os serviços cliente, de forma implícita ou explícita, que os dados

contidos na resposta podem ou não serem armazenados em cache. No contexto da utilização do protocolo HTTP na implementação de uma aplicação REST, a indicação da possibilidade de cache pode acontecer de forma implícita através do método utilizado na requisição, por exemplo, requisições *GET* costumam ser entendidas como passiveis de armazenamento em cache por padrão. Por outro lado, requisições *POST* costumam não ser passiveis de armazenamento em cache. Além disso, o controle do cache pode ser feito de forma explícita através da utilização de cabeçalhos HTTP (*headers*) nas respostas dadas aos clientes, por exemplo, *Expires*, *Cache-Control* e *ETag*.

A possibilidade de armazenar as respostas do servidor em cache traz a capacidade de aumentar a qualidade do serviço sendo oferecido pela aplicação REST. A utilização de cache pode reduzir a latência percebida na comunicação entre cliente e servidor, bem como reduzir a carga de trabalho realizada pelos servidores (FIELDING, 2000).

2.3.4 Interface uniforme

A restrição de que a interface da aplicação REST deve ser uniforme é descrita por Fielding (2000) como uma das restrições fundamentais para simplificar a arquitetura do sistema e aumentar a visibilidade das interações que são possíveis através da interface. Através da uniformidade da interface, as implementações são desacopladas dos serviços fornecidos, o que se torna um fator importante na capacidade de evolução deles. Entretanto, a generalização da interface para torná-la uniforme pode trazer reduções de desempenho para algumas aplicações que se beneficiariam de uma representação mais especializada para suas necessidades.

Para atingir uma interface uniforme, Fielding (2000) descreve algumas restrições que podem servir de guias. Primeiramente, as requisições devem identificar os recursos sendo utilizados na interação e para isso podem utilizar *Uniform Resource Identifiers* (URIs). Além disso, a representação dos recursos contidos nas respostas do servidor devem conter informações suficientes para serem manipuláveis. Por fim, os clientes devem receber informações sobre como manipular os recursos e referências (através de links) para outros recursos que podem ser necessários para completar uma tarefa.

2.3.5 Sistema baseado em camadas

Fielding (2000) descreve que essa restrição permite que aplicações REST sejam compostas por camadas hierárquicas de forma que clientes possam se conectar com componentes intermediários entre a aplicação cliente e servidora e, ainda assim, receber as respostas do servidor. Isto é feito de forma que tais camadas não fiquem aparentes para as aplicações cliente. Por exemplo, a aplicação pode ser pensada para executar em múltiplos servidores com múltiplas camadas de seguranças, autorização, regras de negócio e cache. Tais camadas podem ser auxiliar no controle e manutenção de cache e balanceamento de carga entre os múltiplos servidores.

Uma arquitetura como esta pode aumentar significativamente a escalabilidade do sistema APAGAR sem adicionar complexidade para as aplicações clientes, visto que desconhecem a estrutura intermediária entre cliente e servidor (AMAZON, 2022; FIELDING, 2000). Entretanto, a adição de múltiplas camadas pode resultar em uma maior latência entre cliente e servidor devido ao *overhead* das camadas intermediárias, o que poderia diminuir a desempenho percebida pelo cliente (FIELDING, 2000).

2.3.6 Código sob demanda

Código sob demanda diz respeito a possibilidade da aplicação servidora enviar código executável para o cliente com a finalidade de estender ou customizar as funcionalidades oferecidas pela aplicação cliente (REDHAT, 2020). Fielding (2000) descreve essa restrição como opcional.

2.3.7 Recursos

Conforme indicado por Fielding (2000), em REST, recursos são a representação primária dos dados e informações da aplicação. Qualquer informação que possa ser nomeada de alguma forma pode ser um recurso na aplicação e a representação deste recurso define o estado dele naquele momento. A representação consiste nos dados pertencentes ao recurso, possíveis metadados e referências para outros recursos. Tais recursos possuem identificadores, os quais utilizados para realizar referenciá-lo durante manipulações em seu estado. Exemplos de recursos são documentos ou imagens, e objetos não virtuais

como pessoas e lugares.

Outro conceito importante relacionados aos recursos em REST são os métodos de recursos. Estes métodos são utilizados para realizar alterações no estado do recurso, por exemplo, atualizar a idade de um recurso que represente uma pessoa (FIELDING, 2000). Dado que REST costuma ser utilizado em conjunto com o protocolo HTTP, métodos de recursos comumente utilizados são os próprios métodos disponibilizados pelo protocolo HTTP com alguma semântica aplicada para o contexto, por exemplo, método *GET* para acessar o recurso, *PUT* para atualizar, *DELETE* para deletar (AMAZON, 2022). Entretanto, é importante ressaltar que REST não necessariamente precisa utilizar o protocolo HTTP e que caso o utilize, não é necessário adotar a semântica comumente utilizada para os métodos. O ponto mais importante é que a interface seja uniforme, independente do protocolo ou semântica aplicada (FIELDING, 2000).

2.4 *Serverless*

Serverless computing, ou computação sem servidores, é um modelo de computação em nuvem com conceitos diferentes daqueles que são habitualmente aplicados em *cloud computing*. No modelo de *serverless computing* os detalhes da infraestrutura e plataforma na qual os serviços são executados ficam escondidos dos clientes, de forma que a atenção dos clientes fica voltada exclusivamente para as funcionalidades desejadas para aplicação, enquanto o restante das responsabilidades são gerenciadas pelo provedor do serviço (SHAFIEI; KHONSARI, 2019; HASSAN; BARAKAT; SARHAN, 2021). Grandes empresas do ramo de *cloud computing*, como Google, Amazon e Microsoft, oferecem serviços *serverless* e, embora, as plataformas possam possuir diferenças, todas buscam entregar um modelo de computação *pay-as-you-go* com capacidades de escalonamento automático (*auto-scaling*) de forma acessível (SHAFIEI; KHONSARI, 2019).

Nesse modelo computação em nuvem, Shafiei and Khonsari (2019) indicam que sua adoção possui três objetivos principais. Primeiramente, deseja-se remover a responsabilidade de lidar com detalhes de infraestrutura ou plataformas, de forma que essa responsabilidade seja empregada pelo provedor de forma transparente para os clientes. Em seguida, busca-se um modelo de cobrança baseado em *pay-as-you-go*, ou seja, o cliente pagará apenas por aqueles recursos que foram utilizados. Por fim, os serviços são escalonados automaticamente de acordo com a demanda dos clientes. Ao atingir tais objetivos, obtêm-se serviços verdadeiramente *serverless*, nos quais toda infraestrutura é gerenci-

ada pelo provedor, o cliente pagará apenas pelo que foi utilizado e suas aplicações serão rapidamente escaladas em resposta de picos de utilização.

A entidade básica dos serviços *serverless* oferecidos por provedores costuma ser funções. Através de eventos ou requisições de usuários, as funções registradas pelos clientes são invocadas para realizar algum processamento e, então, retornar os resultados do processamento. A invocação das funções é realizada por algum dos nós de computação disponíveis na infraestrutura própria do provedor do serviço, os quais costumam ser *containers Docker* ou algum ambiente isolado de computação (SHAFIEI; KHONSARI, 2019).

Embora Shafiei and Khonsari (2019) indiquem que não exista uma definição formal do conceito de *serverless computing*, os autores definiram algumas categorias desse modelo de computação, as quais não ficam limitadas apenas à oferta das entidades básicas: funções. Tais categorias são:

- ***Function as a service* (FaaS):** modelo computação *serverless* em que o cliente desenvolve, executa e gerencia as funcionalidades desejadas para sua aplicação através da utilização de funções sem a necessidade de construir e manter infraestrutura para isso, além dos demais benefícios do modelo discutidos anteriormente;
- ***Backend as a Service* (BaaS):** um serviço online é oferecido como uma solução para lidar com tarefas específicas, por exemplo, autenticação e notificações. Os recursos necessários para este serviço são completamente gerenciados pela provedora. Este modelo difere-se da categoria FaaS, pois é oferecido um serviço completo, sem a necessidade do desenvolvimento completo de uma aplicação;
- ***Serverless service*:** trata-se de uma generalização das categorias FaaS e BaaS. Tais serviços devem conter características as características de *serverless computing* descritas anteriormente e costumam ser compostas de duas partes: (1) uma aplicação cliente que implementa grande parte da lógica do serviço e que interage com o usuário final do serviço e o provedor, de forma que invoque funções registradas pelo usuário de um lado e converta os resultados para o formato adequado do provedor; e (2) funções registradas no provedor, as quais invocadas por eventos pré-definidos.

Os serviços *serverless* descritos trazem benefícios que reduzem a complexidade de implantação (*deploy*) e manutenção de aplicações, escalabilidade automática de forma acessível e, possivelmente, reduções de custos financeiros atrelados a *cloud computing* (SHAFIEI; KHONSARI, 2019; HASSAN; BARAKAT; SARHAN, 2021). Estas vanta-

gens são vistas como oportunidades para o desenvolvimento de aplicações em tempo-real ou voltadas para processamento de vídeo, inteligência artificial, Internet das coisas, e-commerce, entre outas. Entretanto, estes benefícios são acompanhados de alguns desafios (SHAFIEI; KHONSARI, 2019).

Dado que *serverless computing* é um modelo relativamente novo, ferramentas para auxiliar no desenvolvimento e *frameworks* podem não ser suficientemente completos, o que pode reduzir a qualidade do código e, consequentemente, a manutenibilidade da aplicação devido à ausência de uma abordagem de desenvolvimento clara (SHAFIEI; KHONSARI, 2019). O mesmo pode ser aplicado para tarefas de *debugging* e teste das aplicações. Além disso, o grande número de provedores no mercado, os quais possuem preços e modelos de cobrança com pequenas variações, podem reduzir significativamente a previsibilidade dos custos financeiros de operações, em especial dado que os modelos são baseados na utilização de recursos (SHAFIEI; KHONSARI, 2019; HASSAN; BARAKAT; SARHAN, 2021).

Outro desafio que deve ser considerado ao utilizar um modelo *serverless* é a comunicação entre funções através da rede. Normalmente, aplicações *serverless* são compostas por múltiplas funções, as quais precisam interagir para completarem tarefas. Em abordagens tradicionais de *cloud computing* isso é abordado através do endereçamento da rede. Entretanto, isso pode se tornar desafiador em um ambiente *serverless* devido à natureza do escalonamento automático, o qual pode fazer com que em um dado momento um grande número invocações das funções estejam executando em diferentes locais do planeta. Além disso, funções costumam possuir um tempo de vida curto, pois são encerradas após ficarem ociosas. Isso requer com que o sistema de endereçamento seja rápido o suficiente para acompanhar as rápidas alterações na estrutura da rede (SHAFIEI; KHONSARI, 2019; HASSAN; BARAKAT; SARHAN, 2021).

Embora a capacidade de escalar para zero o número de instâncias em momentos de inatividade seja normalmente vista como um grande benefício do modelo *serverless*, essa característica acompanha outro desafio importante de ser avaliado: o *cold start* (ASLANPOUR et al., 2021). O tempo de *cold start* refere-se ao tempo necessário para inicialização de um nova instância. Neste processo, a instância precisa ser carregada juntamente com suas dependências e inicializada, o que adiciona uma latência de dezenas ou centenas de milissegundos para que a função fique pronta para processar requisições. Para algumas aplicações, a latência adicionada devido ao *cold start* pode ser impeditiva. Nesses casos, faz-se necessário buscar alternativas e estratégias para reduzir ou mitigar

a latência de *cold starts* (ASLANPOUR et al., 2021; HASSAN; BARAKAT; SARHAN, 2021).

Além dos desafios citados, Shafiei and Khonsari (2019) indicam que aplicações *serverless* também compartilham desafios com aplicações de *cloud computing* tradicionais. Estratégias e soluções de cache local ou distribuído também devem ser consideradas, assim como questões de segurança, privacidade e autenticação.

2.5 *Infrastructure as Code* (IaC)

No contexto da implementação do código de aplicações, Mansoor et al. (2020) indicam que regras precisam ser seguidas para que uma aplicação seja válida, para isso utilizam-se formatos e sintaxes específicos. Ferramentas de gerenciamento e versionamento de código são aplicadas para controlar o histórico de desenvolvimento, correções e alterações do *software*. Assim, ao adotar a prática de *Infrastructure as Code* (IaC), busca-se utilizar o mesmo rigor aplicado ao código de aplicações para o provisionamento de infraestrutura.

Logo, *Infrastructure as Code* (IaC) trata-se do processo de gestão e provisionamento da infraestrutura de sistemas através da utilização de código, ao invés de processos manuais (REDHAT, 2022). Todas configurações aplicadas a infraestrutura devem ser definidas através de código e armazenadas em uma ferramenta de versionamento de código. Através da utilização desta abordagem, o provisionamento, a orquestração e a implantação da infraestrutura são realizados através de código. Tradicionalmente, esses processos eram feitos através de processos manuais e com o auxílio de *scripts*, o que facilmente pode trazer problemas quando alterações são realizadas na infraestrutura e não são documentadas (MANSOOR et al., 2020; REDHAT, 2022).

A utilização de IaC evita que alterações na infraestrutura sejam feitas sem serem documentadas no histórico de alterações. Com a utilização de arquivos contendo a especificação da infraestrutura, o processo de edição e distribuição das configurações é facilitado e obtém-se a garantia de que o mesmo ambiente será provisionado todas às vezes (REDHAT, 2022). Além disso, a velocidade de implantação de *software* é aumentada em conjunto com a consistência da infraestrutura, enquanto que os custos financeiros das operações e a quantidade de erros são reduzidos (MANSOOR et al., 2020; REDHAT, 2022).

A especificação da infraestrutura através da utilização de código pode ser realizada

forma declarativa ou imperativa. Na abordagem declarativa define-se o estado desejado para o sistema e sua infraestrutura. Já na abordagem imperativa, definem-se os comandos que levam a infraestrutura a atingir o estado desejado. Exemplos de ferramentas de IaC são: Terraform[1], Amazon CloudFormation[2] e Puppet[3] (REDHAT, 2022).

[1]<https://www.terraform.io/>
[2]<https://aws.amazon.com/cloudformation/>
[3]<https://www.puppet.com/>

3 TECNOLOGIAS UTILIZADAS

Neste Capítulo, apresentamos as principais tecnologias utilizadas no desenvolvimento do sistema de notificações proposto por este trabalho, bem como as motivações para sua escolha. Visamos descrever as tecnologias, seus casos de usos, vantagens e desvantagens, além de realizar comparações com tecnologias com propósitos semelhantes. O objetivo deste detalhamento é apresentar as características das tecnologias utilizadas para fundamentar a escolha delas. O nível de detalhes apresentado em cada tecnologia é diretamente proporcional a importância desta no desenvolvimento do trabalho e no embasamento desejado para justificar a escolha.

3.1 *Serverless Framework*

Serverless Framework é uma ferramenta *open-source* para desenvolvimento de aplicações *serverless* que consiste em uma interface de linha de comando (CLI) e um painel de controle e gerenciamento. Através desta ferramenta pode-se gerenciar o ciclo de vida de desenvolvimento de aplicações *serverless* por completo. A complexidade do processo de desenvolvimento, implantação, segurança e solução de problemas é facilitado pela utilização do *framework*. Diferentemente de outros *frameworks*, este auxilia no gerenciamento do código da aplicação e da infraestrutura. Além disso, o *framework* dá suporte a diversas linguagens de programação (SERVERLESS-FRAMEWORK, 2023).

O *Serverless Framework* abstrai diversos serviços *serverless* dos principais provedores de serviços em nuvem do mercado. Através dessa abstração é possível descrever sua infraestrutura e realizar a implantação em qualquer um dos provedores suportados. Entretanto, o *framework* não impede a utilização de outras ferramentas de IaC em conjunto com as abstrações fornecidas, o que garante a aplicação de todas configurações disponíveis pelos provedores, por exemplo, no caso da AWS, *templates* de CloudFormation podem ser utilizados em conjunto com as definições do *framework* (SERVERLESS-FRAMEWORK, 2023).

Existem quatro conceitos principais no *framework*, os quais são brevemente descritos a seguir:

- ***Functions*:** representa as funções *serverless*, conforme descritas na categoria FaaS anteriormente, as quais são implantadas no provedor como uma unidade de execu-

ção independente. Uma função é simplesmente a porção de código utilizada para exercer uma tarefa específica. Ao utilizar a AWS como *cloud provider*, as funções são implantadas no serviço AWS Lambda;

- ***Events***: as funções desenvolvidas e implantadas são invocadas através de eventos, os quais podem ser originados de uma ação de um usuário ou por outro serviço, por exemplo, um sistema de fila e sistemas de mensageira. Os eventos que invocam uma função são especificados através de código declarativo seguindo a abstração fornecida pelo *framework*;
- ***Resources***: são componentes de infraestrutura presentes no provedor de serviços em nuvem, por exemplo, um banco de dados e uma fila distribuída. Estes recursos podem ser declarados tanto através das abstrações do *framework* ou através da sintaxe adequada para o provedor;
- ***Services***: serviços são simplesmente unidades de organização utilizadas pelo *framework* para agrupar funções, eventos e recursos relacionados.

As aplicações e a infraestrutura associada a elas são definidas através da utilização dos conceitos citados. Eles podem ser declarados através de arquivos JSON, YAML, *Javascript* e *Typescript*. A partir dessas declarações, de IaC e das integrações com provedores de serviços em nuvem que ocorre a implantação e provisionamento(SERVERLESS-FRAMEWORK, 2023).

Além disso, o *Serverless Framework* permite que suas funcionalidades sejam estendidas através da implementação de *plugins*, os quais são muito úteis para auxiliar no desenvolvimento e gerenciamento da aplicação, bem como em tarefas relacionadas a testes e *debugging*.

A escolha do *Serverless Framework* como ferramenta para organizar a aplicação *serverless* desenvolvida neste trabalho e a infraestrutura relacionada foi fortemente baseada na capacidade de aceleração da implementação. As abstrações presentes no *framework* permitem que os desenvolvedores evoluam rapidamente no desenvolvimento sem a necessidade de configurar previamente todo o ambiente *serverless*. Além disso, o grande número de *plugins* disponíveis para utilização em conjunto com o *framework* aumentam ainda mais o poder da ferramenta, por exemplo, ao facilitar tarefas complexas em ambientes *serverless*, como *debugging* e testes. Por se tratar de uma ferramenta altamente difundida no mercado, ela possui uma grande e engajada comunidade de desenvolvedores, o que significa uma grande variedade de conteúdos disponíveis para consulta.

3.2 AWS Lambda

AWS Lambda é um serviço de computação *serverless* baseado em funções, o qual oferece alta escalabilidade e permite a execução de código sem a necessidade de provisionar e gerenciar infraestrutura (AMAZON, 2023j). O serviço funciona como um tipo de *Function as a Service* (FaaS), o qual realiza a administração de todos os recursos de computação, servidores, sistemas operacionais, manutenções, escalonamento automático e *logging*. Nele é possível desenvolver praticamente qualquer tipo de aplicação em qualquer uma das linguagens suportadas pelo serviço (BAIRD et al., 2017).

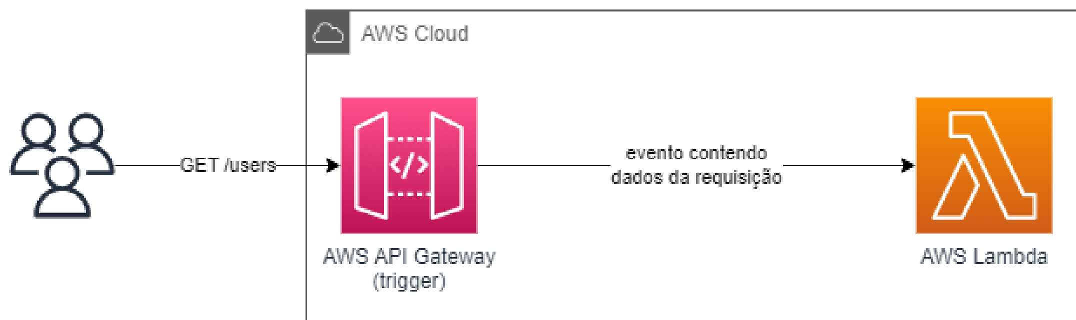
As funções Lambda podem ser invocadas por uma série de eventos de outros serviços da AWS. Essa característica permite que sejam desenvolvidas aplicações reativas e baseadas em eventos. Quando múltiplos eventos são recebidos simultaneamente, de forma que eventos são recebidos mais rapidamente do que podem ser processados, o AWS Lambda invoca novas instâncias das funções para os eventos serem processados paralelamente. Quando tráfego é reduzido, as instâncias ociosas são paradas ou congeladas. Dessa forma, o cliente só é cobrado nos períodos em que as funções estão inicializando ou processando eventos (BAIRD et al., 2017; AMAZON, 2023i).

As entidades que invocam a execução de funções Lambda são chamadas de gatilhos, ou *triggers*. Estes gatilhos podem ser outros recursos e serviços da AWS, os quais podem ser configurados para invocarem determinadas funções Lambda na ocorrência de determinados eventos. Eventos são documentos no formato JSON, os quais contém os dados necessários para a função Lambda executar (AMAZON, 2023i). Na Figura 3.1, pode ser visualizado o serviço da Amazon API Gateway atuando como *trigger* de uma função Lambda, quando uma nova requisição (evento) é realizada por usuários da API. Um evento contendo os dados da requisição é passado para a função Lambda configurada para ser invocada em resposta a ele.

A escolha de utilizar o serviço AWS Lambda está altamente relacionada à utilização de uma arquitetura *serverless* para o desenvolvimento deste trabalho. Optou-se por utilizar uma arquitetura *serverless* devido à natureza de um sistema de notificações e a capacidade de auxiliar nos desafios relacionados a este tipo de sistema.

Conforme discutido na seção 2.2, um sistema de notificações, assim como muitos outros tipos de sistema, terá sua utilização baseada em grandes momentos de picos de uso e vários momentos de inatividade. Ao possuir diversas aplicações de *software* integradas ao sistema notificações, prever os picos de uso do sistema torna-se uma tarefa complexa,

Figura 3.1: Amazon API Gateway atuando como *trigger* de uma função Lambda.



visto que cada aplicação terá necessidades diferentes em períodos do dia diferentes. Além disso, o sistema de notificações deve ser capaz de suprir o volume de notificações decorrente de possíveis grandes aumentos da base de usuários das aplicações integradas. Estes picos podem ser ainda maiores dado que clientes do sistema de notificações podem, simultaneamente, necessitar grandes quantidades de envios de notificações.

Um sistema de notificações amplamente baseado na utilização de serviços *serverless*, como FaaS, pode fornecer a capacidade de escala necessária para atender seus clientes nos picos de utilização e ainda reduzir os custos de operação. Dada a capacidade que serviços FaaS possuem de rapidamente aumentar o número de instâncias disponíveis a medida que mais requisições são recebidas, um sistema de notificações baseado neste serviço *serverless* terá a escalabilidade necessária para atender a imprevisibilidade dos picos de utilização. Ao mesmo tempo, devido à natureza do sistema descrito, em momentos de baixa utilização ou completa inatividade, o número de instâncias sofrerá grandes reduções, o que garante grandes reduções nos custos financeiros. Logo, o poder de escalabilidade e a redução de custos de operação foram os principais motivadores para a escolha desta abordagem.

Além disso, ao adotar uma abordagem *serverless*, o sistema de notificações adquire as características citadas de forma simples e totalmente gerenciada. Isto elimina a necessidade de realizar a manutenção de servidores e permite que os desenvolvedores passem mais tempo focados no desenvolvimento de novas funcionalidades, aumentando a velocidade de desenvolvimento.

Entretanto, o *cold start* deve ser considerado ao avaliar a abordagem *serverless*. Para um sistema de notificações com as tecnologias utilizadas neste trabalho, as funções são pequenas e coesas, o que faz com que o tempo de *cold start* seja na ordem de poucas centenas de milissegundos. Dado que um sistema de notificações não exige tempos de

resposta como aplicações críticas de IoT, a adição de uma possível latência de algumas centenas de milissegundos no envio de uma notificação não é nada impeditivo. Por isso, o sistema não exige que grandes otimizações do tempo de *cold start* sejam realizadas.

Por fim, a escolha pelo serviço AWS Lambda como provedor de FaaS foi baseada no grande ecossistema *serverless* da AWS, o qual permite a integração entre diversos serviços *cloud* de maneira muito simples e prática. Além disso, experiências prévias com o ambiente *cloud* da AWS foram decisivos para preferência de seus serviços, ao invés dos serviços fornecidos por outros *cloud providers*.

3.3 Amazon *Simple Queue Service* (SQS)

Amazon *Simple Queue Service* (SQS) é um serviço de fila distribuída totalmente gerenciado que permite o desacoplamento e a escala de aplicações *serverless*, microserviços e sistemas distribuídos. Por ser totalmente gerenciado, o SQS elimina a complexidade de gerenciar a infraestrutura de um sistema de filas dedicado, o que permite aos desenvolvedores focarem apenas no desenvolvimento de soluções. A utilização do SQS permite que mensagens sejam enviadas, armazenadas e recebidas entre aplicações para qualquer necessidade de escala e volume. Além disso, as mensagens enviadas ao SQS são persistidas, o que garante que mensagens enviadas não sejam perdidas (AMAZON, 2023f).

Este serviço oferece dois tipos de filas gerenciadas: *standard* e *FIFO*. As filas *standard* oferecem o máximo *throughput* e garantem uma estratégia de entrega *at-least-once*, enquanto a estratégia de ordenamento aplicada é *best-effort*. Logo, devido à natureza altamente distribuída que permite o altíssimo *throughput*, ocasionalmente, uma mesma mensagem pode ser entregue duas vezes e é possível que o ordenamento das mensagens não corresponda a ordem em que foram adicionadas à fila. Já as filas FIFO possuem funcionalidades extras para garantir que não existam mensagens duplicadas e que as mensagens sejam entregues com uma estratégia de *exactly-once processing*. Assim, através da utilização estratégias de deduplicação por conteúdo ou identificador, garante-se que não existirão mensagens duplicadas em janelas de 5 minutos. Entretanto, essas garantias significam que o *throughput* da fila é reduzido em comparação com filas *standard* (AMAZON, 2023f).

Tais características do SQS trazem benefícios como a durabilidade das mensagens, alta disponibilidade e segurança (as mensagens podem ser encriptadas pelo próprio

serviço). Além disso, o SQS irá escalar automaticamente durante aumentos ou picos de utilização, bem como aumentará a confiabilidade no processamento, visto que *locks* são aplicados nas mensagens ao serem consumidas para garantir que não existirão múltiplos consumidores processando a mesma mensagem. Todas essas características são acessíveis através de um modelo de cobrança *pay-as-you-go* baseado na quantidade de requisições realizadas para o serviço (AMAZON, 2023f).

Optou-se por utilizar SQS como uma forma de adicionar uma camada extra de durabilidade entre o SNS e as funções do AWS Lambda. O SQS permite que sejam definidas regras de retenção de mensagens, número de tentativas de processamento e estratégias de reprocessamento de mensagens que falharam via *dead letter queues*. Assim, ao adicionar essa camada de durabilidade, evita-se que mensagens sejam perdidas sem chance de reprocessamento e, essa é uma garantia altamente desejada para um sistema de notificações, visto que perder mensagens pode significar o não envio de uma notificação. Além disso, ao adicionar o SQS em frente às funções do AWS Lambda, temos uma espécie de *"buffer"* de mensagens, o que permite que estratégias de *throttling* sejam exploradas no processamento das mensagens como forma de aliviar a pressão em recursos utilizados pelas funções Lambdas.

Outra vantagem motivadora para preferência pelo SQS, ao invés de outros serviços de fila distribuída disponíveis no mercado, foi o fato de ser completamente gerenciado e possuir fácil integração com outros serviços disponíveis na AWS. Isso elimina diversas tarefas que estariam associadas ao provisionamento e manutenção da infraestrutura de serviços semelhantes. Ao mesmo tempo, ao optar por este serviço, reduções de custos financeiros poderão ser observadas, dado que o sistema só é cobrado por aquilo for utilizado de fato.

3.4 Amazon *Simple Notification Service* (SNS)

O Amazon *Simple Notification Service* (SNS) é um serviço de entrega de mensagens de *publishers* para *subscribers* (*pub/sub* ou produtores e consumidores) distribuído e completamente gerenciado. No SNS, produtores se comunicam com consumidores de forma assíncrona através de da publicação de mensagens em tópicos. Os consumidores, por sua vez, se inscrevem para receber mensagens de tópicos, o que garante que todos os consumidores inscritos em um tópico recebem as mensagens destinadas a ele (AMAZON, 2023j; AMAZON, 2023g).

O SNS oferece alta disponibilidade, segurança e um esquema de escalabilidade automática que garante o *throughput* necessário em momentos de altos níveis de utilização. Semelhante ao SQS, o SNS possui dois tipos de tópicos: *standard* e *FIFO*. Ao optar por tópicos *FIFO*, é necessário utilizar filas SQS FIFO como consumidores dos tópicos (AMAZON, 2023g). Além disso, tópicos *FIFO* possuem maiores limitações no número máximo de consumidores por tópico (AMAZON, 2023e). Por fim, o acesso ao SNS é feito por meio de um modelo de cobrança baseado no número de mensagens enviadas e recebidas (AMAZON, 2023g).

Assim, como no caso do SQS, a escolha de utilizar o SNS foi fortemente baseada na facilidade de integração e no fato de ser completamente gerenciada e oferecer um modelo de cobrança *pay-as-you-go*. Além disso, priorizou-se a utilização do SNS, ao invés de serviços semelhantes, como o EventBridge, pois o SNS é um serviço mais simples e que permite um *throughput* maior, bem como um número maior de *subscribers*.

3.5 Amazon DynamoDB

O Amazon DynamoDB é um serviço de banco de dados NoSQL, o qual oferece alta e consistente desempenho em qualquer escala e volume de dados. O serviço surgiu da necessidade de um banco de dados com alta escalabilidade e disponibilidade que pudesse atender as demandas das aplicações Amazon. Para isso, o desenvolvimento foi guiado em torno dos requisitos de construir um banco dedados com desempenho consiste, alta disponibilidade, durabilidade e totalmente gerenciado e *serverless*. A alta capacidade do DynamoDB foi demonstrada em 2021 durante um evento de 66 horas da Amazon, no qual as aplicações da Amazon realizaram trilhões de requisições para a API do DynamoDB. Um pico de 89,2 milhões de requisições por segundo foi atingido, enquanto o serviço mantinha a alta disponibilidade com desempenho de apenas um dígito de milissegundo (ELHEMALI et al., 2022).

Para atingir tais marcas, o DynamoDB armazena os dados em tabelas, as quais são uma coleção de itens e, por sua vez, os itens são uma coleção de atributos. Os itens presentes em uma tabela são unicamente identificados a partir de uma chave primária. As chaves primárias são compostas obrigatoriamente por uma chave de partição (*partition key*) e opcionalmente por uma chave de ordenamento (*sort key*). Múltiplos itens podem possuir a mesma *partition key*, porém só é permitido um item por par de *partition key* e *sort key*. A *partition key* é utilizada para indicar em qual partição interna do DynamoDB o

item será armazenado, já a *sort key* é utilizada para ordenar os itens que possuem a mesma *partition key* (ELHEMALI et al., 2022; AMAZON, 2023c).

Além disso, as tabelas do DynamoDB possuem múltiplas partições e são utilizadas para garantir os requisitos de *throughput* e armazenamento, por isso os usuários do banco de dados devem definir as *partition keys* de maneira que os itens sejam distribuídos adequadamente (ELHEMALI et al., 2022; AMAZON, 2023c). Para garantir a alta disponibilidade e durabilidade dos itens, as partições são replicadas em diferentes zonas de disponibilidade dentro da infraestrutura da AWS (ELHEMALI et al., 2022).

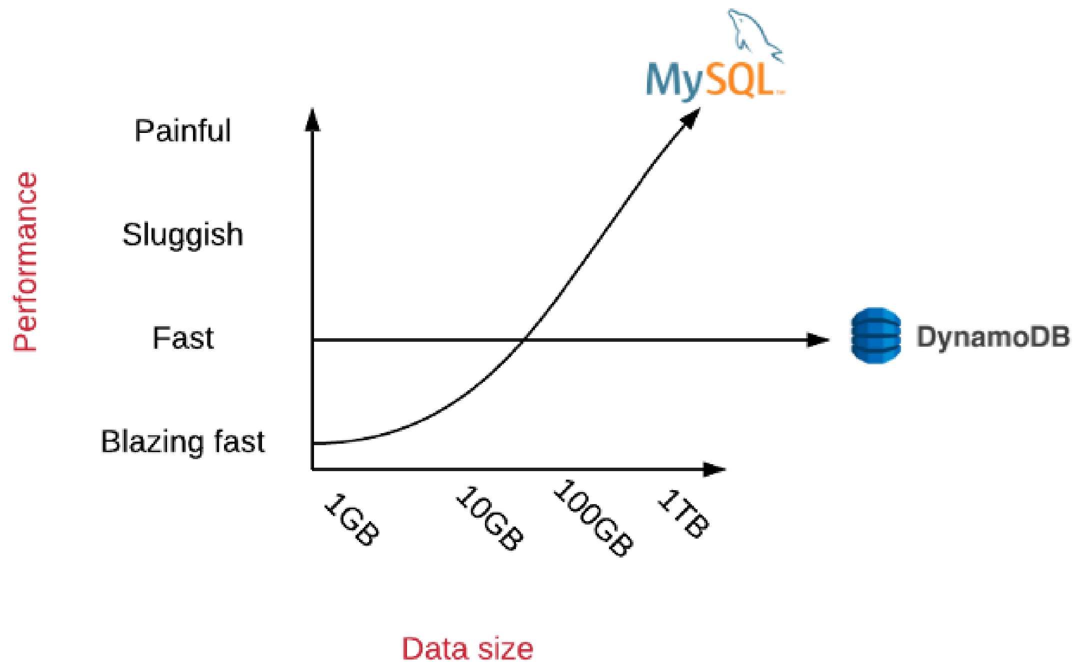
O acesso aos itens contidos em tabelas é feito a partir das chaves primárias, o que é utilizado para garantir a consistência no desempenho do DynamoDB. A fim de aumentar a capacidade de consulta dos dados, o DynamoDB possui suporte a índices secundários, os quais podem ser utilizados para realizar consultas a partir de outros atributos, porém isso é acompanhado do custo de replicação dos dados dos itens (ELHEMALI et al., 2022; AMAZON, 2023c). Além disso, índices secundários são utilizados apenas para operações de leitura. Toda operação de escrita deve ser realizada com a utilização do índice primário através da chave primária do item. Em relação à escrita, o DynamoDB ainda suporta transações ACID, o que permite que a escrita de múltiplos itens seja feita respeitando a atomicidade, consistência, isolamento e durabilidade dos dados sem penalizar a performance e disponibilidade (ELHEMALI et al., 2022; DEBRIE, 2020).

As operações de leitura são baseadas na utilização de *partition keys* e *sort keys*, mas isso não significa que apenas consultas do tipo chave-valor sejam permitidas. O DynamoDB também permite que consultas sejam feitas por meio de *partition keys* e que opções sejam aplicadas para retornar múltiplos itens contidos na partição especificada conforme o valor das *sort keys* (ELHEMALI et al., 2022; AMAZON, 2023c). Dado que a interface de consulta do DynamoDB é bastante simples e permite poucas formas de acesso aos dados, a escolha dos valores de *partition key* e *sort key* são extremamente importantes e costumam exigir que os padrões de acesso aos dados sejam previamente conhecidos para auxiliar na definição, (DEBRIE, 2020).

Conforme comentado anteriormente, o DynamoDB é um banco de dados *serverless* e totalmente gerenciado com alta escalabilidade. Em conjunto com essas características, o DynamoDB adotou uma abordagem de acesso à API do serviço baseada exclusivamente em requisições HTTP para evitar possíveis gargalos que o gerenciamento de conexões poderia trazer (ELHEMALI et al., 2022; DEBRIE, 2020). Estas e outras características fazem com que o DynamoDB seja uma ótima opção de banco de dados ao

utilizar outros serviços *serverless*, como o AWS Lambda.

Figura 3.2: Comparação da performance de consultas entre DynamoDB e bancos relacionais a medida que o volume de dados armazenado cresce. Fonte: (DEBRIE, 2021)



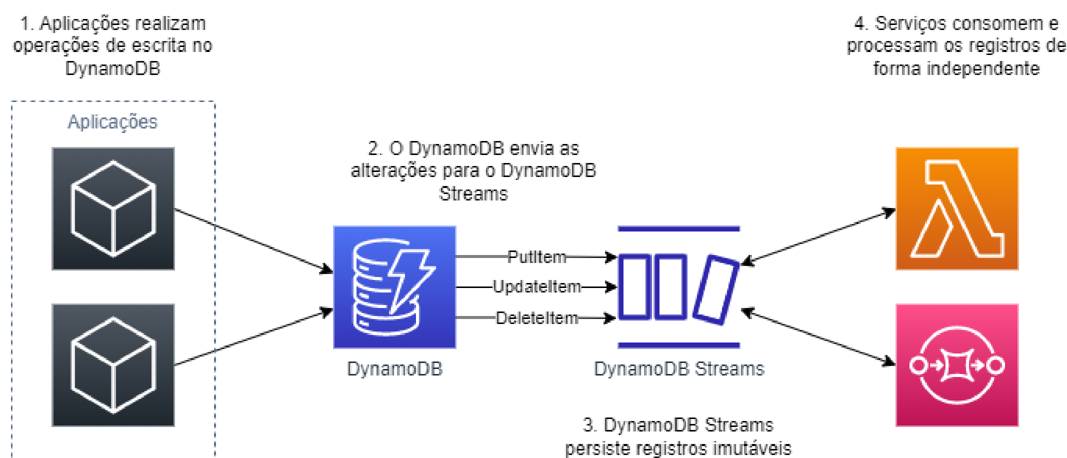
Ao comparar o DynamoDB com bancos relacionais, uma série de vantagens podem ser observadas. Embora, bancos relacionais permitam uma flexibilidade muito maior do que o DynamoDB e, possivelmente, velocidade de desenvolvimento por serem altamente difundidos, as características de alta escalabilidade e sinergia com aplicações baseadas em FaaS podem ter grande influência na decisão. Segundo DeBrie (2020), escalar bancos relacionais costuma ser um desafio grande e complexo, tornando a escalabilidade e consistência de performance do DynamoDB um ponto bastante relevante. Na Figura 3.2, pode ser visualizado o comportamento comum de bancos relacionais conforme o volume de dados armazenados cresce. Conforme descrito por DeBrie (2021), a medida que mais dados são armazenados, um banco relacional sem estratégias avançadas de escalabilidade pode notar aumentos significativos no tempo necessário para processar consultas, enquanto o DynamoDB mantém constante o tempo necessário para consultas para todos os volumes de dados armazenados.

3.5.1 DynamoDB Streams

Além dos recursos já citados, o DynamoDB possui uma funcionalidade com grande impacto na facilidade de desenvolver aplicações baseadas em eventos. O DynamoDB Streams são sequencias imutáveis de registros contendo alterações realizadas em tabelas do DynamoDB, as quais podem ser processadas por múltiplos consumidores. Essa funcionalidade faz com que tarefas de compartilhamento de dados entre sistemas de forma assíncrona sejam facilmente implementadas (AMAZON, 2023c; DEBRIE, 2020).

Com o DynamoDB Streams é possível escolher entre receber através do *stream* de inserções, atualizações e remoções apenas as chaves primárias afetadas, o item antes da alteração, o item depois da alteração ou ambas imagens do item. Além disso, é possível adicionar filtros ao *stream* para que apenas algumas alterações sejam disponibilizadas no DynamoDB Streams (AMAZON, 2023c). De acordo com DeBrie (2020), isso permite que diversos casos de uso sejam desenvolvidos, visto que há suporte para vários serviços da AWS serem utilizados como consumidor do *stream*. Sem dúvida, a integração do DynamoDB Streams com o AWS Lambda garante uma forma totalmente gerenciada e *serverless* de reagir a alterações no banco de dados.

Figura 3.3: Fluxo de disponibilização de alterações realizadas no DynamoDB para o DynamoDB Streams.



Na Figura 3.3 pode ser visualizado o funcionamento do DynamoDB Streams. A medida que aplicações realizam alterações no banco de dados, elas são adicionadas ao DynamoDB Streams e ficam disponíveis para que serviços consumam os registros e reajam às alterações.

3.5.2 Motivação para escolha do DynamoDB

Dentre as opções de banco de dados disponíveis no mercado, optou-se por utilizar o DynamoDB, devido à sinergia que ele possui com o AWS Lambda e os benefícios que suas características podem trazer para um sistema de notificações. Primeiramente, gerenciar conexões de bancos de dados em ambientes *serverless* pode ser uma tarefa complexa e com o DynamoDB esta tarefa é inexistente. Além disso, opções de banco de dados tradicionais, como PostgreSQL, costumam não ser gerenciados, acrescentando tarefas complexas na manutenção da infraestrutura, além de possivelmente se tornarem gargalos de performance.

Em um sistema de notificações, em que há uma grande imprevisibilidade dos picos de uso, a utilização de um banco de dados não gerenciado pode significar a necessidade de manter servidores de bancos de dados com muitos recursos (CPU, memória e armazenamento) para que o sistema consiga suportar o volume de utilização durante os picos. Entretanto, definir a quantidade correta para estes recursos em um ambiente imprevisível pode ser uma tarefa complexa. Além disso, devido à natureza de um sistema de notificações, uma abordagem como esta não seria nada eficiente em termos de custos de operação. Dado que existe a possibilidade de haver vários momentos de baixa utilização do sistema, manter servidores ociosos e de alto custo encarece a operação rapidamente. Logo, utilizar um banco de dados gerenciado com alta capacidade de escala e um modelo de cobrança baseado na utilização do serviço traz uma maior eficiência para o sistema de notificações.

O DynamoDB oferece a alta escalabilidade e desempenho desejada em um modelo completamente gerenciado que contempla os requisitos de um sistema de notificações escalável e eficiente. Além disso, a consistência de performance do DynamoDB garante que a desempenho do sistema de notificações não seja afetado a medida que o volume de dados armazenados aumenta. Portanto, essas características foram decisivas para a utilização do DynamoDB como banco de dados principal.

Além das motivações já citadas, o DynamoDB Streams também foi avaliado durante a tomada de decisão, visto que este recurso permite que o sistema de notificações reaja a alterações realizadas no banco de dados de forma muito simples e performática, sem a necessidade de utilizar serviços adicionais para esta tarefa, como o *Debezium* para bancos relacionais.

3.6 Amazon EventBridge Scheduler

O Amazon EventBridge Scheduler é um serviço *serverless* dedicado para realizar tarefas relacionadas a agendamentos. Este é um serviço gerenciado que permite a criação, execução e gerenciamento de tarefas agendadas. Ele foi construído para ser altamente escalável, de forma que permita o agendamento de milhões de tarefas, as quais podem ser processadas por uma enorme lista de serviços da AWS. Por se tratar de um serviço *serverless* e completamente gerenciado, não há a necessidade de realizar o gerenciamento ou provisionamento de nenhuma infraestrutura. O serviço pode ser utilizado para agendamento de tarefas recorrentes por meio de expressões *cron* e expressões de recorrência usando uma estrutura simples. Agendamentos com execução única também são possíveis através da especificação da data e hora desejada. A natureza *serverless* do serviço permite que o modelo de cobrança adotado seja baseado no número de agendamentos realizados, o que garante reduções de custo quando comparado a soluções que exigem gerenciamento de infraestrutura dedicada (AMAZON, 2023d).

Optou-se por utilizar o Amazon EventBridge Scheduler, ao invés de serviços semelhantes, como o Amazon CloudWatch Events, pois ele possui recursos adicionais e capacidades de customização. Além disso, o Amazon EventBridge Scheduler possui a possibilidade de integração com diversos serviços da AWS e de terceiros.

3.7 Amazon API Gateway

O Amazon API Gateway também é um serviço totalmente gerenciado, o qual permite que APIs sejam criadas para exercerem o papel de porta de entrada para aplicações. Isso permite que usuários acessem dados, regras de negócio e funcionalidades de serviços *back-end* conectados ao API Gateway, por exemplo, funções do AWS Lambda. A solução permite que APIs REST, HTTP e WebSocket sejam criadas, publicadas, gerenciadas, monitoradas e protegidas na escala de centenas de milhares de requisições concorrentes. O acesso a esse serviço é feito a partir de um modelo de cobrança baseado no número de requisições recebidas (ou conexões, para APIs WebSocket), assim, além de não ser necessário gerenciar nenhuma infraestrutura, paga-se apenas o que foi realmente utilizado do serviço (AMAZON, 2023a; AMAZON, 2023j).

Neste serviço, APIs criadas na categoria HTTP são compostas por coleções de rotas e métodos, os quais são integrados a outros serviços responsáveis por processar as

requisições, por exemplo, funções do AWS Lambda. As APIs criadas na categoria REST possuem grande parte das funcionalidades de APIs HTTP, porém com a adição de funcionalidades avançadas, como validação do corpo de requisições e estratégias de *throttling* por cliente. Além disso, diferentemente das APIs HTTP, essas APIs são organizadas em coleções de recursos e métodos. Devido ao acesso a recursos avançados, as APIs REST possuem um custo maior por requisição maior do que APIs HTTP. Por fim, APIs WebSocket, como o nome sugere, utilizam o protocolo WebSocket e são organizadas em conjuntos de rotas e chaves (AMAZON, 2023a).

Um conceito importante do Amazon API Gateway são os *Lambda Authorizers*, também chamados de autorizadores personalizados. Esta funcionalidade permite que funções do AWS Lambda sejam desenvolvidas e utilizadas para controlar o acesso à API através de esquemas de autorização especializados para as necessidades da aplicação (AMAZON, 2023a; DEBRIE, 2019).

Dada a necessidade de expor funções do AWS Lambda por meio de uma API REST, a utilização do API Gateway para gerenciar as rotas expostas torna-se natural, visto que este é o serviço padrão da AWS para realizar a tarefa de gerenciar e escalar APIs. Além de possuir uma integração simples com o serviço do AWS Lambda, o API Gateway fornece recursos de segurança para a API. A escolha da utilização do serviço de API REST do API Gateway, ao invés do API HTTP, foi baseada na possibilidade utilização de recursos avançados para o gerenciamento da API, os quais são bastante úteis e não estão presentes no serviço de API HTTP.

3.8 Amazon Cognito

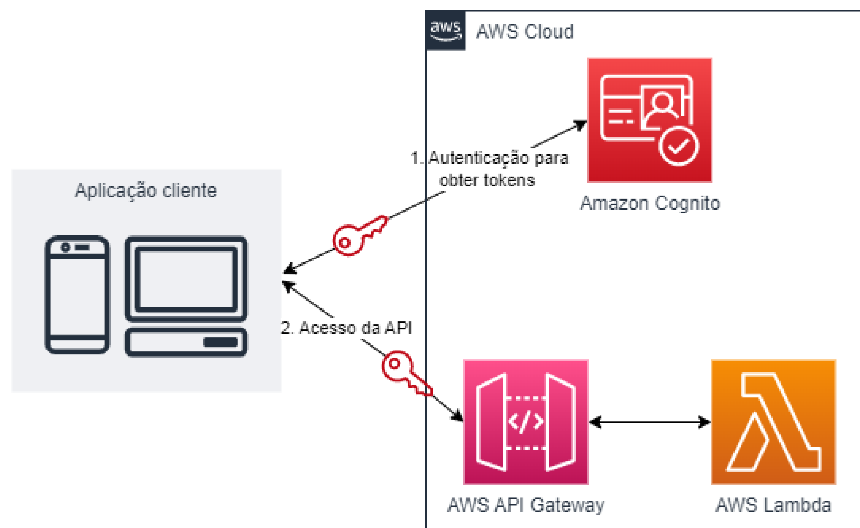
O Amazon Cognito é um serviço que permite a autenticação, autorização e gerenciamento de usuários para aplicações *web* e *mobile*. O serviço oferece uma ferramenta completa, a qual permite que recursos de cadastro e acesso às contas sejam adicionados às aplicações com agilidade, o que evita a necessidade de desenvolver uma solução própria para esta tarefa. Ademais, por se tratar de um serviço gerenciado, não há a necessidade gerenciar infraestrutura extra e a solução pode escalar para milhões de usuários enquanto oferece o acesso à aplicação através de usuário e senha ou provedores de identidade como Facebook, Twitter, Apple, entre outros, bem como fluxos personalizados (AMAZON, 2023j; AMAZON, 2023b).

Este serviço possui dois componentes principais: *users pools* e *identity pools*. O

primeiro permite que usuários se cadastrem e acessem aplicações próprias. Já o segundo é voltado para que usuários da aplicação possam acessar outros serviços da AWS. Estes componentes podem ser utilizados em conjunto ou não, e permitem a implementação de diversos casos de uso (AMAZON, 2023b).

Um caso de uso bastante comum e que é explorado por este trabalho é o acesso a recursos da aplicação através do Amazon API Gateway e AWS Lambda com a utilização do Amazon Cognito com uma *user pool* (AMAZON, 2023b). Na Figura 3.4 é possível visualizar o fluxo de acesso à aplicação com a utilização do Amazon Cognito como entidade autorizadora. Neste caso de uso, os usuários utilizam o Amazon Cognito para obter *tokens* de acesso, os quais são utilizados para autenticar as requisições enviadas para o Amazon API Gateway. Ao receber tais *tokens* no cabeçalho das requisições, o API Gateway utiliza o Cognito como entidade autorizadora para validar os *tokens* e garantir o acesso aos recursos oferecidos pela aplicação através do API Gateway.

Figura 3.4: Acesso a recursos da aplicação através do Amazon API Gateway e AWS Lambda com a utilização do Amazon Cognito com uma *user pool*.



O Amazon Cognito foi escolhido como serviço de gerenciamento de autenticação, autorização e usuários, pois, além de ser totalmente gerenciado e possuir um sistema de cobrança *pay-as-you-go*, ele possui integração simplificada com os demais serviços da AWS.

3.9 AWS CloudFormation

O AWS CloudFormation auxilia na declaração e configuração de recursos AWS através de IaC, o que reduz o tempo consumido em atividades de gerenciamento de serviços da AWS. Por meio de *templates* contendo a descrição e configuração desejada dos serviços e recursos da AWS, o CloudFormation realiza o provisionamento e configuração dos serviços. Assim, a necessidade de realizar a criação manual de cada um dos recursos deixa de existir, bem como a necessidade de avaliar dependências entre recursos (AMAZON, 2023j; AMAZON, 2023h).

Existem três conceitos importantes do Amazon CloudFormation: *templates*, *stacks* e *change sets*. O primeiro diz respeito ao código utilizado para declarar os recursos desejados através da sintaxe de *templates* do CloudFormation em arquivos JSON ou YAML. Já o segundo trata-se do agrupamento de recursos relacionados (descritos por *templates*) em uma unidade chamada de *stack*, a qual permite gerenciar os recursos presentes na unidade de forma isolada. Por fim, quando deseja-se realizar alterações nos recursos presentes em uma *stack*, gera-se um *change set* antes de concretizar as alterações para que o impacto da implementação delas seja avaliado (AMAZON, 2023h).

A decisão de utilizar o AWS CloudFormation para provisionar e implantar a infraestrutura da aplicação foi baseada exclusivamente na facilidade de integração do *Serverless Framework* com o serviço e seus *templates*.

3.10 PlanetScale e Vitess

PlanetScale é um serviço de banco de dados *serverless* compatível com MySQL (PLANETSCALE, 2023). Este serviço é construído como uma camada de abstração para o Vitess, uma solução para gerenciamento de instâncias de bancos de dados (PLANETSCALE, 2023; VITESS, 2023a). Nesta seção, estes serviços são descritos, bem como suas capacidades e vantagens.

3.10.1 Vitess

Vitess é uma solução de banco de dados voltada para a implantação, escala e gerenciamento de *clusters* com um grande número de instâncias de bancos de dados. Ela

foi criada em 2010 por engenheiros do YouTube para solucionar problemas de escalabilidade, após outras soluções tradicionais não terem sido suficientes. Após a introdução do Vitess como um componente entre a aplicação e o banco de dados, foi possível remover código da aplicação voltado para o tratamento das múltiplas instâncias, além de auxiliar o YouTube a escalar sua base de usuários por um fator maior do que 50 vezes (VITESS, 2023b). O sucesso da solução fez com que ela fosse adotada por grandes empresas, como Slack[1], Pinterest[2] e GitHub[3] (VITESS, 2023a).

Esta solução é compatível com MySQL e *Percona Server* para MySQL, de forma que combina e aprimora várias funcionalidades de bancos relacionais com a escalabilidade de bancos NoSQL. Assim, alguns dos objetivos do Vitess são: (1) escalar bancos de dados facilmente mediante estratégias de *sharding* com o mínimo de alterações necessárias no código da aplicação, (2) facilitar a migração de bancos de dados em servidores dedicados ou máquinas virtuais para infraestruturas de nuvem privada ou pública, e (3) facilitar a implantação e gerenciamento de *clusters* de bancos de dados (VITESS, 2023a).

Ao comparar Vitess com a utilização de MySQL em sua forma tradicional, algumas diferenças ficam evidentes. Cada conexão de um banco de dados MySQL utiliza de 256 KB até 3 MB de memória, isso significa que escalar as aplicações que utilizam o banco de dados, o aumento no número conexões exige mais memória, o que pode aumentar consideravelmente o custo de operação (VITESS, 2023a). No caso do Vitess, cada conexão exige uma quantidade de memória bem menor, de maneira que o sistema de *pooling* de conexões se beneficie da alta capacidade de concorrência de Go (linguagem em que é construído) para mapear tais conexões para um conjunto menor de conexões MySQL, o que garante a capacidade de gerenciar milhares de conexões facilmente (VITESS, 2023a; MORRISON, 2022). Enquanto MySQL não possui nenhum esquema de *sharding* nativamente, o Vitess possibilita diversos esquemas, os quais permitem aumentar e diminuir de *shards* de forma pouco intrusiva. Além disso, Vitess possui várias funcionalidades para aumentar performance e evitar consultas problemáticas, por exemplo, consultas simultâneas e idênticas passam por processo de deduplicação para poder reutilizar o resultado da consulta em execução (VITESS, 2023a).

Já ao comparar o Vitess com bancos NoSQL, alguns benefícios podem ser identificados. Por executar acima de instâncias de MySQL, as consultas podem utilizar o poder do SQL para realizar buscas complexas com a utilização de *JOINs*, agregações e demais

[1]<https://slack.com/>
[2]<https://pinterest.com/>
[3]<https://github.com/>

recursos. Vitess também dá suporte a transações, as quais não estão presentes em várias alternativas NoSQL (VITESS, 2023a).

Entretanto, para permitir a alta escalabilidade e poder de *shardings*, o Vitess introduz algumas incompatibilidades com o MySQL, por exemplo, a ausência do conceito de chaves estrangeiras (VITESS, 2023a). Essas incompatibilidades fazem com que algumas responsabilidades tenham que ser implementadas no código da aplicação (PLANETSCALE, 2023).

3.10.2 PlanetScale

O PlanetScale é um serviço de banco de dados *serverless* compatível com MySQL a partir da utilização do Vitess de maneira totalmente gerenciada. Embora o Vitess introduza uma série de benefícios, a implantação e gerenciamento da infraestrutura necessária para sua utilização são tarefas complexas e que exigem alto conhecimento da ferramenta. Para remover a responsabilidade de administrar a infraestrutura necessária, o PlanetScale fornece um serviço gerenciado que oferece a escalabilidade do Vitess com a facilidade de utilização de serviços *serverless* (MORRISON, 2022; PLANETSCALE, 2023).

Além de tornar acessível à utilização do Vitess, o PlanetScale oferece funcionalidades extras para o gerenciamento do banco de dados. Entre tais funcionalidades, pode-se citar a possibilidade de realizar alterações no *schema* do banco de dados sem ocasionar em bloqueios e *downtime* (PLANETSCALE, 2023). Ademais, por se tratar de um banco *serverless*, a utilização de PlanetScale e Vitess com aplicações altamente baseadas em FaaS é feita de forma bastante natural e sem a necessidade de se preocupar com o limite no número de conexões no banco devido à alta escalabilidade das funções, pois tais soluções suportam mais de um milhão de conexões simultâneas (DIJK, 2022).

3.10.3 Motivação para escolha

Conforme descrito na Seção 3.5, a interface disponibilizada pelo DynamoDB não permite consultas complexas, com várias agregações e filtros aplicados em múltiplos campos. Logo, para permitir que usuários do sistema de notificações visualizem o histórico de envio eficientemente e que estes dados possam ser analisados para trazer informações úteis aos clientes, faz-se necessário buscar alternativas para o armazenamento de infor-

mações. Assim, a escolha do PlanetScale com utilização do Vitess é fortemente apoiada pela necessidade destas consultas complexas. Esta solução foi preferida, ao invés de soluções mais tradicionais como bancos de dados relacionais não gerenciados, justamente por trazer o poder de consulta do SQL em conjunto com uma solução totalmente gerenciada.

A utilização do Vitess através do PlanetScale garante a escalabilidade do sistema de armazenamento e praticamente elimina a preocupação em relação ao número de conexões com o banco, visto que as conexões de baixo custo do Vitess garantem a possibilidade de que milhões de conexões sejam estabelecidas. Além disso, o modelo gerenciado permite a utilização do modelo de cobrança *pay-as-you-go*, o qual possui maior propensão a redução de custos. Por fim, a familiaridade com a linguagem SQL também teve um grande peso na motivação para utilizar o PlanetScale.

3.11 Node.js

Node.js é uma plataforma, ou ambiente de execução, que permite a execução de Javascript em ambientes *server-side*, o que permite o desenvolvimento de aplicações que não executem em um navegador. Para isso, o *runtime* de Node.js foi construído a partir do motor de Javascript utilizado no navegador *Chrome*, o *V8 JavaScript engine*. O propósito da plataforma é permitir a construção de aplicação escaláveis por meio da utilização massiva de operações assíncronas e um modelo I/O baseado em eventos e não bloqueante (NODE.JS, 2022a; NODE.JS, 2022b).

O *V8 JavaScript engine*, que permite a execução do código Javascript, não é apenas um interpretador de código, ele utiliza um esquema de compilação *just-in-time* para tentar compilar porções de código frequentemente utilizadas. Através desse recurso, a execução do código Javascript é acelerada (NODE.JS, 2022b).

O Node.js foi escolhido como ambiente de execução para as funções do AWS Lambda, principalmente em virtude da familiaridade com a tecnologia. Além disso, o amplo ecossistema do Node.js com a utilização de Typescript facilita e acelera o desenvolvimento da aplicação.

4 TRABALHOS RELACIONADOS

Neste Capítulo, descrevemos os principais trabalhos que procuram solucionar a complexidade associada ao desenvolvimento de um sistema de notificações completo. Primeiramente, apresentamos a *Courier*, solução presente a mais tempo no mercado. Na sequência, são apresentadas as soluções construídas por *Knock* e *SuprSend*. Por fim, realiza-se uma análise comparativa entre os trabalhos e suas funcionalidades.

Embora o mercado de notificações possua um grande número de provedores para vários canais diferentes, o número de projetos que buscam solucionar o problema descrito nesse trabalho é bem menor. Conforme é descrito a seguir, esses trabalhos são recentes, o que pode servir como indicativo de que o problema discutido é atual e está em pleno desenvolvimento. Além disso, o recente aumento no uso de notificações e, consequentemente, o aumento do número de opções de provedores no mercado também podem estar relacionados ao surgimento do problema e trabalhos que buscam solucioná-lo.

4.1 *Courier*

A *startup Courier*[1] permite que sua API seja utilizada para criar notificações apenas uma vez e distribuí-las através de qualquer canal suportado. A solução desenvolvida permite que notificações sejam enviadas através de canais, como *SMS*, e-mail, notificações *in-app*, *push notifications* e *chat*. Além disso, o projeto dá suporte a um grande conjunto de provedores para cada canal disponível. O sucesso da *startup* fundada em 2019 permitiu que, em 2022, a rodada de investimentos de série B encerrasse com o valor de US$ 35 milhões (WIGGERS, 2022).

A solução construída pela *Courier* também oferece recursos mais avançados além do envio de notificações através de múltiplos canais. Usuários sem conhecimento técnico podem utilizar um *studio* de edição da *Courier* para construir o *design* de suas notificações e criar *templates* para todos os canais disponíveis. Os *templates* criados podem ser gerenciados e utilizados para o envio de notificações e, assim, diminuir a complexidade das requisições para a API. Além disso, os *templates* podem ser construídos utilizando blocos de conteúdo definidos pelo *studio* e que abstraem a complexidade do formato adequado para cada canal.

Também é permitido que os usuários customizem algumas regras de roteamento

[1] <https://www.courier.com/> Acesso em Setembro de 2022

pré-estabelecidas pelo sistema para cada notificação criada. Porém, para criar fluxos de envio com roteamento completamente personalizável para os requisitos de seus usuários, a *Courier* permite a construção de fluxos baseados em etapas através de uma interface de usuário. As etapas disponíveis permitem ações, como enviar notificação, interromper a execução por tempo determinado, iniciar outro fluxo de envio, atualizar perfil do destinatário e buscar informações em APIs externas para compor o conteúdo das notificações.

Já em relação ao controle de preferências de destinatários, a empresa fornece uma solução baseada em categorias, as quais podem ser associadas a *templates* de notificações. Entretanto, este método possui limitações nas possibilidades de personalização e controle de preferências por parte do destinatário. Por isso, a *Courier* estuda um novo mecanismo de preferências baseado em tópicos de notificações, os quais permitem ao destinatário escolher seus interesses e os canais desejados. Entretanto, este mecanismo estava em fase de testes até a data do último acesso.

Por fim, toda notificação enviada através do sistema é registrada e seus dados são disponibilizados para consulta através de um histórico de envio. Entretanto, nenhum tipo de análise é feita sobre estes dados para extrair conhecimento sobre o comportamento dos destinatários. A *startup* utiliza uma abordagem cobrança *pay-per-use*, ou seja, um valor é cobrado por cada notificação enviada através do sistema.

4.2 *Knock*

O projeto *Knock*[2], fundado em 2021, se descreve como "infraestrutura de notificações para desenvolvedores" e visa facilitar o envio de notificações por meio de múltiplos canais por meio de uma API simples e diversas funcionalidades. O projeto oferece suporte para vários provedores de canais, como *SMS*, e-mail, notificações *in-app*, *push notifications* e *chat*. Além de fornecer a infraestrutura de um sistema de notificações, a solução possui uma série de funcionalidades que buscam melhorar o envio de notificações. O acesso a essas funcionalidades é feito mediante uma estratégia de cobrança baseada em categorias conforme o volume de notificações enviadas, ou seja, o número de notificações enviadas no período de um mês definem o valor da cobrança e a categoria que o cliente se encaixa.

O *Knock* ainda oferece a possibilidade de criação e gerenciamento de *templates* reutilizáveis voltados para o canal de e-mail. Para editar estes *templates*, a solução fornece

[2]<https://knock.app/> Acesso em Setembro de 2022

um editor de notificações baseado exclusivamente em texto, o qual permite que *templates* sejam construídos utilizando HTML ou *markdown*. Não há a possibilidade de definir e reutilizar *templates* para outros canais, como SMS e *push notifications*. Além disso, as mensagens para os demais canais suportados também precisam ser definidas utilizando um editor de texto, o que impede que *templates* sejam construídos de forma que os detalhes de cada canal sejam abstraídos, bem como impede que usuários sem conhecimento técnico definam seus próprios *templates*

Além disso, a solução fornecida pelo *Knock* requer que seus usuários criem fluxos personalizados de envio de notificações para ser possível realizar qualquer envio de notificações. Isto significa que a API provida não permite que notificações sejam enviadas isoladamente e sem a necessidade prévia de construir através da interface de usuário. Entretanto, a construção dos fluxos personalizados permite a utilização de diversos recursos avançados para a criação de estratégias de roteamento completas. Estes fluxos são construídos a partir de sequências de etapas de execução, as quais permitem funcionalidades, como o envio de notificação através dos canais suportados, a interrupção da execução do fluxo por tempo determinado e a busca de informações em APIs externas para compor o conteúdo das notificações.

Já o controle de preferências de destinatários é realizado através de um modelo complexo, o qual permite que preferências sejam associadas a fluxos de envio ou categorias de notificações. O sistema permite que o destinatário escolha quais canais mais adequados para cada fluxo de envio definido pela empresa. Entretanto, a gestão das configurações associadas aos grupos de preferências existentes é feita majoritariamente através da API do *Knock*, o que impediria um membro sem conhecimento técnico gerir as preferências existentes.

Finalmente, as notificações enviadas através do sistema são registradas e apresentadas num formato de histórico que permite análise de desempenho no envio. O *Knock* não extrai conhecimento destes dados, mas apresenta métricas básicas, como o número de notificações abertas e lidas. Além disso, o *Knock* fornece a funcionalidade de *batch* de mensagens para que destinatários não sejam sobrecarregados com mensagens em uma curta janela de tempo. Assim, define-se uma duração para janela e todas as solicitações de envio recebidas para um destinatário são agrupadas e enviadas ao final da janela.

4.3 *SuprSend*

O projeto *SuprSend*[3], lançado ao público em 2022, visa oferecer uma infraestrutura confiável e escalável para produtos de *software* que desejam enviar notificações sem precisar construir e dar manutenção para uma solução proprietária. *SuprSend* oferece suporte para aos principais canais e seus provedores, como *SMS*, e-mail, notificações *in-app*, *push notifications* e *chat*. Para acessar as funcionalidades da solução, a estratégia de cobrança utilizada pelo projeto é baseada em categorias conforme o volume de notificações enviadas, assim como para o projeto *Knock*.

Da mesma maneira que *Courier* oferece a possibilidade de criação e gerenciamento de *templates* através de um editor que abstrai os detalhes de cada canal, o *SuprSend* possui uma solução própria para *templating* com controle de versionamento. Os *templates* criados a partir do editor ficam disponíveis para servirem como base no envio de notificações para os múltiplos canais e provedores disponíveis na solução. Além disso, estratégias de roteamento são aplicadas para as notificações enviadas através do sistema, porém, possibilidades de personalização das estratégias não são disponibilizadas para os desenvolvedores. O gerenciamento de preferências dos destinatários também não é suportado pelo produto da *SuprSend*. Assim, cabe aos desenvolvedores controlarem quais destinatários desejam ou não receber as notificações. Além disso, assim como o *Knock*, a solução do *SuprSend* permite o envio de notificações em *batch* para reduzir a quantidade de notificações recebidas por um destinatário em curtos intervalos de tempo.

Por fim, toda notificação enviada através do sistema é disponibilizada por meio de um histórico. Entretanto, o histórico fornecido pelo *SuprSend* não possui informações detalhadas sobre as informações enviadas para o provedor, as repostas obtidas e detalhes de possíveis erros. O conjunto de dados gerados durante o envio de notificações e interações com destinatários é utilizado para apresentar dados analíticos que auxiliam as organizações a entender o comportamento de seus destinatários e notificações. Os relatórios disponíveis apresentam informações como: qual o canal e provedor com o melhor desempenho e interação com destinatários, quais *templates* apresentam resultados melhores, o quão engajado os destinatários estão e taxas de visualização por destinatário.

[3]<https://www.suprsend.com/> Acesso em Setembro de 2022

4.4 Análise comparativa

A partir da análise das três soluções citadas, percebe-se que elas possuem um conjunto de recursos bastante semelhante. Porém, a abordagem utilizada por elas e o foco da solução possuem diferenças. Enquanto a *Courier* e o *SuprSend* parecem focar também em uma experiência amigável para membros sem conhecimento técnico por meio de editores amigáveis e interfaces simples, o *Knock* foca na experiência do desenvolvedor que utiliza o sistema de notificações. Além disso, o *SuprSend*, a solução menos madura entre as observadas, tem seu foco voltado para casos de uso mais básicos e que não requerem muitas personalizações e recursos. Todas as soluções suportam os mesmos canais de distribuição de notificações, mas a quantidade total de provedores disponíveis é bem maior para os usuários de *Courier*. Essas disparidades podem estar relacionadas a diferença de maturidade dos projetos, visto que o *Knock* e, principalmente, o *SuprSend* são projetos bem mais recentes, embora os três possuam algumas semelhanças no conjunto de funcionalidades oferecidas.

Apesar dos três projetos terem como objetivo principal solucionar o mesmo problema, isto é, facilitar o envio de notificações por meio de múltiplos canais e provedores, a forma como buscam alcançá-lo é diferente. Essa diferença pode ser notada ao verificar a documentação da API de cada uma delas. O fluxo de envio de notificações é bastante diferente e, com certeza, isso reflete em diferenças na forma como os projetos alcançam o objetivo. Além disso, a forma como abordam cada funcionalidade também influencia na diferença do público alvo de cada solução.

Outra diferença importante é a possibilidade de roteamento de notificações. *Courier* e *Knock* oferecem soluções avançadas e poderosas para o roteamento de notificações através da construção de fluxos de envio personalizados, embora a abordagem de envio da *Courier* seja mais flexível e permita o envio sem a necessidade de definir fluxos personalizados. Já o *SuprSend* não possui nenhuma forma de configurar ou personalizar o roteamento aplicado.

As estratégias de *templating* adotadas por cada uma das soluções também é diferente. Enquanto *Courier* e *SuprSend* possuem capacidades de gerenciamento de *templates* reutilizáveis para qualquer canal, o *Knock* limita esta funcionalidade apenas para e-mail. Além disso, o *Knock* não possui nenhum tipo de abstração no editor de *templates*, restringindo a criação deles aos membros da equipe técnica. Por outro lado, *Courier* e *SuprSend* fornecem um editor simples com campos previamente definidos e blocos de

"arrastar e soltar". Diferentemente das outras soluções, a *Courier* ainda abstrai o conteúdo das notificações através de conjuntos de blocos de conteúdo, os quais facilitam o compartilhamento de conteúdo entre *templates* de diferentes canais.

As diferenças citadas também são refletidas no modelo de cobrança, dado que os projetos utilizam abordagens diferentes, embora todas estejam associadas ao volume de notificações. Com o amadurecimento das soluções citadas, é esperado que novos projetos concorrentes surjam para simplificar o envio de notificações através de outras abordagens e agreguem o que há de melhor em cada uma.

Com base nas descrições dadas para cada trabalho neste Capítulo e na análise comparativa realizada nesta Seção, podemos agregar suas principais funcionalidades e características na Tabela 4.1. O conteúdo desta Tabela comparativa auxiliou a guiar o desenvolvimento do sistema de notificações desenvolvido neste trabalho, visto que permite a visão clara das principais funcionalidades presentes em trabalhos semelhantes já existentes no mercado.

Tabela 4.1: Análise comparativa das funcionalidades e características presentes nas soluções de *Courier*, *Knock* e *SuprSend*.

Funcionalidade ou característica	***Courier***	***Knock***	***SuprSend***
Abstrai canais e provedores	Sim	Sim	Sim
Suporta e-mail	Sim	Sim	Sim
Suporta SMS	Sim	Sim	Sim
Suporta *in-app*	Sim	Sim	Sim
Suporta *chat*	Sim	Sim	Sim
Suporta *push notifications*	Sim	Sim	Sim
Número total de provedores	48	23	13
Envio em *batch*	Não	Sim	Não
Roteamento de notificações	Sim	Sim	Sim
Fluxos avançados de roteamento	Sim	Sim	Não
Gerenciamento de *templates*	Sim	Sim[4]	Sim
Editor de *templates* amigável	Sim	Não	Sim
Conteúdo abstraído em blocos	Sim	Não	Não
Preferências de destinatários	Sim	Sim	Não
Histórico de envio	Sim	Sim	Sim
Histórico detalhado e com erros	Sim	Sim	Não
Análise de performance	Não	Não	Sim
Modelo de cobrança	*Pay-per-use*	Categorias	Categorias

Além dos projetos analisados em mais detalhes anteriormente, é relevante citar a existência de provedores que, embora fornecem o serviço de notificações em múltiplos canais, não fornecem uma interface única e simplificada para enviar notificações. Um

[4] O *Knock só permite gerenciamento de templates para e-mails.*

exemplo de provedor com essas características é o *Twilio*[5], o qual possui uma capitalização de mercado de, aproximadamente, 12 bilhões de dólares (TWILIO, 2022). O projeto possui soluções para envio de notificações através de e-mail, *Whatsapp*, SMS e outros. Porém, para cada um dos canais oferecidos, há uma API diferente para desenvolvedores se integrarem e enviarem notificações. Esta característica permite que desenvolvedores explorem todos recursos disponíveis para o canal, porém dificulta a disponibilização de notificações em múltiplos canais devido a desunião das APIs. Além disso, por se tratar de um provedor, embora exista a possibilidade de utilizar múltiplos canais do serviço, as integrações desenvolvidas ficam acopladas ao provedor, o que impede a adição de suporte a múltiplos provedores sem a necessidade de alterar o código de integração com o serviço de notificações.

[5] <https://www.twilio.com/> Acesso em Setembro de 2022

5 DESENVOLVIMENTO E IMPLEMENTAÇÃO

Neste Capítulo, a implementação do sistema de notificações exemplificado na Figura 2.1 é descrita em detalhes. Para isso são identificadas as tecnologias utilizadas em cada módulo do sistema, a comunicação entre os serviços que o compõem e as funções presentes em cada um deles, bem como as rotas expostas pela API REST implementada por cada módulo. Em cada Seção deste Capítulo, buscou-se organizar o conteúdo de forma que seja apresentado primeiramente a visão geral de cada conceito e, posteriormente, detalhes são acrescentados incrementalmente. Assim, ao final do Capítulo deseja-se que o funcionamento do sistema de notificações e o fluxo de envio sejam completamente compreendidos, bem como as diferenças e semelhanças do sistema com os trabalhos relacionados.

5.1 Visão Geral

O sistema de notificações descrito na Seção 2.2 e exemplificado na Figura 2.1 foi construído a partir de três módulos distintos, os quais são referenciados como serviços ou *services*. Estes serviços são brevemente descritos a seguir:

- ***Identity and Access Management* (IAM) *Service*:** serviço responsável por gerenciar os usuários (organizações e empresas de *software*) do sistema de notificação, bem como o acesso destes usuários aos demais serviços do sistema;
- ***Notification Service*:** este serviço é responsável por gerenciar as entidades envolvidas no envio de notificações e executar todos os processos associados ao envio através de integrações com provedores;
- ***Analytics Service*:** serviço responsável por coletar e armazenar todas as informações geradas durante o envio de notificações e disponibilizá-las de forma que facilite a consulta detalhada, bem como permitir que análises sejam feitas análises sobre os dados coletados para extrair informações úteis.

A comunicação entre esses três principais serviços é realizada de forma completamente assíncrona através da utilização do AWS SNS e SQS. Cada um dos serviços emite, por meio de tópicos *standard* do SNS, eventos e mensagens relevantes em relação aos fatos ocorridos durante o processamento. Isso permite que os serviços interessados possam se inscrever nos tópicos para receber eventos e reagir a eles da forma adequada

para a responsabilidade do serviço. Assim, os serviços atuam como *publishers* e *subscribers* de tópicos do SNS. Além disso, é comum que filas SQS sejam utilizadas como camada adicional de durabilidade entre as funções do serviço e o SNS visando usufruir das características citadas na Seção 3.4.

Dado que os serviços utilizam intensamente a comunicação assíncrona e que erros podem ocorrer durante a comunicação e o processamento dos dados, é importante que estratégias sejam adotadas para mitigar o efeito destes erros. No sistema de notificações implementado, utilizaram-se os mecanismos de retentativa de execução das funções AWS Lambda, o que faz com que a própria AWS seja responsável por tentar executar novamente uma mensagem que falhou anteriormente. O mesmo também foi aplicado para o SQS, o qual foi configurado para as mensagens poderem ser retentadas algumas vezes até que elas sejam movidas automaticamente para *dead-letter queues*. Este tipo de fila é importante, pois nos casos em que, mesmo após retentar, o processamento da mensagem falha, ela é persistida e uma notificação da infraestrutura é recebida para as falhas serem averiguadas. Assim, após o problema ser solucionado, a mensagem pode ser reinserida na fila de origem e processada corretamente, evitando que mensagens sejam perdidas devido a falhas no sistema. É importante ressaltar que nos diagramas apresentados neste Capítulo as filas de *dead-letter queue* foram omitidas para simplificar a representação, porém toda fila SQS possui uma fila de *dead-letter queue*.

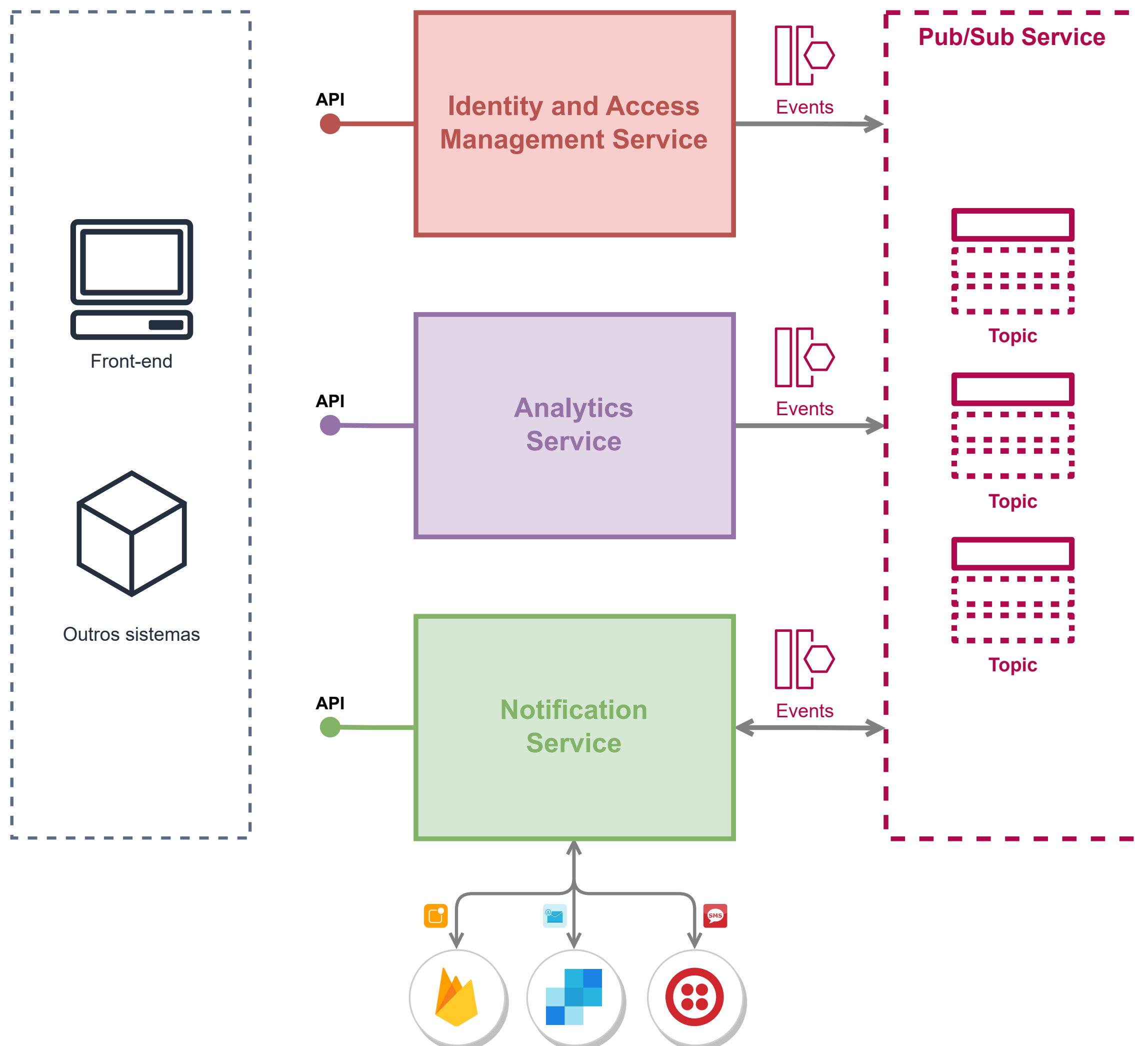
Além disso, cada um dos serviços descritos também implementa uma API REST, a qual é exposta na Internet através do serviço Amazon API Gateway. Estas APIs permitem que clientes realizem o gerenciamento do sistema de notificações e seus destinatários, o monitoramento dos envios realizados pelo sistema e, obviamente, o envio de notificações. As rotas que compõem estas APIs REST são voltadas para dois casos de uso: aplicação *front-end* e outros *softwares* que utilizem o sistema de notificações.

As APIs voltadas para a aplicação *front-end* foram construídas para serem utilizadas pelos usuários do sistema de notificações (empresas de *software* e desenvolvedores) através de uma interface de usuário destinada ao gerenciamento do sistema de notificações. Já as APIs voltadas para outros *softwares* foram construídos para facilitar a tarefa de integrar-se ao sistema de notificações para realizar o envio de e-mails, *push notifications* e SMS.

Logo, o sistema de notificações construído pode ser visualmente representado como na Figura 5.1. Três serviços que utilizam tópicos do SNS para realizarem a comunicação assíncrona entre eles e que expõem APIs REST, as quais podem ser utilizadas

por uma aplicação *front-end* e múltiplos *softwares*. Além disso, o *Notification Service* realiza comunicação síncrona e assíncrona com as APIs de provedores para permitir o envio de notificações.

Figura 5.1: Visão geral do sistema de notificações e os serviços que o compõem.



5.2 Provisionamento e implantação

Os processos de provisionamento de implantação serviço foram totalmente realizadas através da integração do *Serverless Framework* com o AWS Cloud Formation. Através das declarações da infraestrutura do sistema feitas por meio das abstrações do *framework* e de *templates* do AWS Cloud Formation, utiliza-se o comando do *Serverless Framework* a seguir para empacotar e realizar o *upload* do código da aplicação. Ao mesmo tempo, inicia-se a execução da *stack* do AWS Cloud Formation que será respon-

sável por configurar a infraestrutura.

```
serverless deploy --stage dev --region us-east-1
```

No comando é possível especificar para qual ambiente deseja-se realizar à implantação, por exemplo, ambiente de desenvolvimento, testes ou produção. Também é possível especificar a região geográfica em que o sistema de será implantado. Neste trabalho, optou-se por utilizar a região *us-east-1*, a qual é conhecida por ser a região de menor custo da AWS.

5.3 Padronizações da API

Dado que cada um dos serviços presentes no sistema de notificações possui uma API REST, utilizamos algumas padronizações para manter a consistência na interface que a API disponibiliza para seus usuários. Essa estratégia foi adotada com o objetivo de diminuir a curva de aprendizagem necessária para compreender o funcionamento da API e torná-la mais intuitiva.

Para isso, padronizamos o formato do caminho das rotas para conterem os recursos da API manipulados por elas, bem como a semântica dos métodos aplicados em cada rota. Por exemplo, uma rota que acesse as preferências de um destinatário é nomeada de maneira que agrupe estes recursos da seguinte forma:

```
/recipients/:recipientId/preferences
```

Já em relação aos métodos aplicados nas rotas, o método HTTP `GET` foi utilizado para consultas, `POST` para inserções e criações de recursos, `PUT` para atualizações e criações e, por fim, `DELETE` para deletar recursos existentes.

Além disso, o formato de entrada e saída de dados também foi padronizado. Para isso, utilizou-se o formato de JSON, no qual os campos sempre foram nomeados com o padrão *camelCase* para manter consistência e evitar que os usuários tenham que lidar com a adição de complexidade da falta de previsibilidade dos nomes dos campos. Para valores do tipo *enum*, ou seja, em que há uma lista pré-definida de possíveis valores para um campo, sempre adotou-se *strings* com todos os caracteres em minusculo, por exemplo, um *enum* de canais de distribuição: `sms`, `push`, `email`.

O formato das respostas de erro também foram padronizadas, para que usuários da API consigam tratá-los da forma adequada. Assim, toda resposta de erro possui um campo

chamado `statusCode` para indicar o código de resposta HTTP, um campo `message` contendo uma mensagem descritiva do erro e, opcionalmente, um campo chamado `details`, o qual contém detalhes sobre o erro, por exemplo, em caso de erro de validação, os detalhes sobre os campos que falharam são informados neste objeto.

Através destas padronizações, buscou-se melhorar a experiência do desenvolvedor (DX) que utiliza a API do sistema de notificações e, consequentemente, facilitar o desenvolvimento de integrações.

5.4 Módulo de *Identity and Access Management* (IAM) *Service*

Esta Seção tem como objetivo descrever em detalhes o funcionamento do módulo de *Identity and Access Management* (IAM). Para isso, são apresentadas a visão geral do funcionamento do serviço, bem como os detalhes de sua arquitetura e API REST.

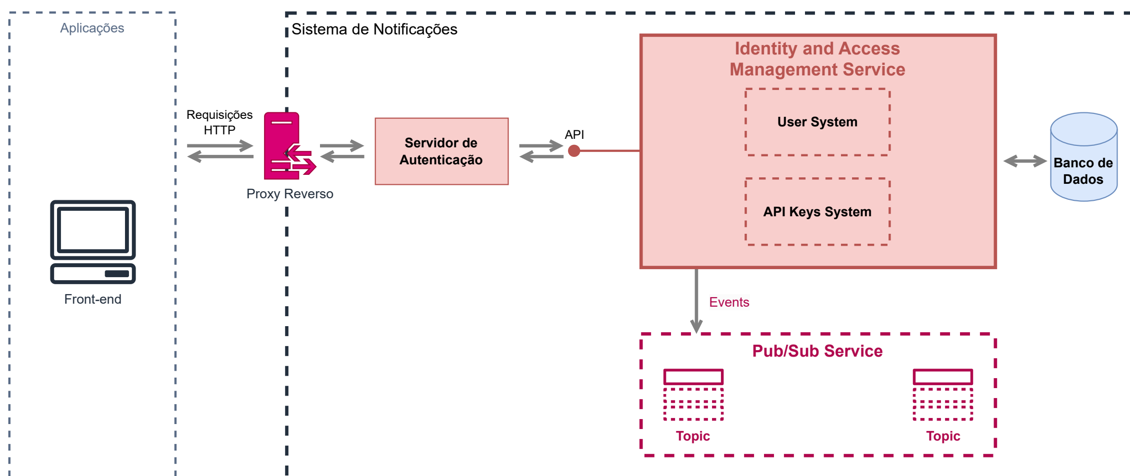
5.4.1 Visão Geral

O serviço de *Identity and Access Management* (IAM) é responsável exclusivamente por gerenciar os usuários do sistema de notificações e as chaves de acesso ao sistema. O cadastro de novos usuários é realizado através deste serviço, bem como a autenticação e autorização. Com a construção deste serviço, remove-se a responsabilidade de gerenciar estes detalhes nos demais serviços que compõem o sistema de notificação.

Com isso, podemos definir a arquitetura de alto nível da Figura 5.2. Através dela podemos representar os principais componentes do sistema de notificações necessários para o funcionamento do serviço. Consequentemente, identifica-se a necessidade de um *proxy* reverso para receber e rotear as requisições de aplicações para o serviço, como o Amazon API Gateway e NGINX. Também, é necessário autenticar usuários e autorizar suas requisições e, para isso, um servidor de autenticação pode ser usado, como Keycloak, Auth0 e Amazon Cognito. Na figura ele foi representado em vermelho, assim como o próprio serviço de IAM, pois este servidor pode fazer parte do serviço ou ser um sistema externo utilizado para essa finalidade. Além disso, necessita-se de um sistema de mensageria ou *pub/sub* distribuído para que os eventos atrelados aos usuários do sistema sejam emitidos para os demais serviços presentes no sistema de notificações. Para isso, podemos utilizar o Kafka, Google Cloud Pub/Sub ou AWS SNS e AWS SQS. Por fim,

um banco de dados será necessário para persistir os dados de usuários e configurações de acesso e, neste caso, dependendo da abordagem utilizada no desenvolvimento, diferentes soluções podem ser utilizadas, como Amazon DynamoDB, MySQL ou MongoDB.

Figura 5.2: Arquitetura de alto nível do serviço de *Identity and Access Management* do sistema de notificações.



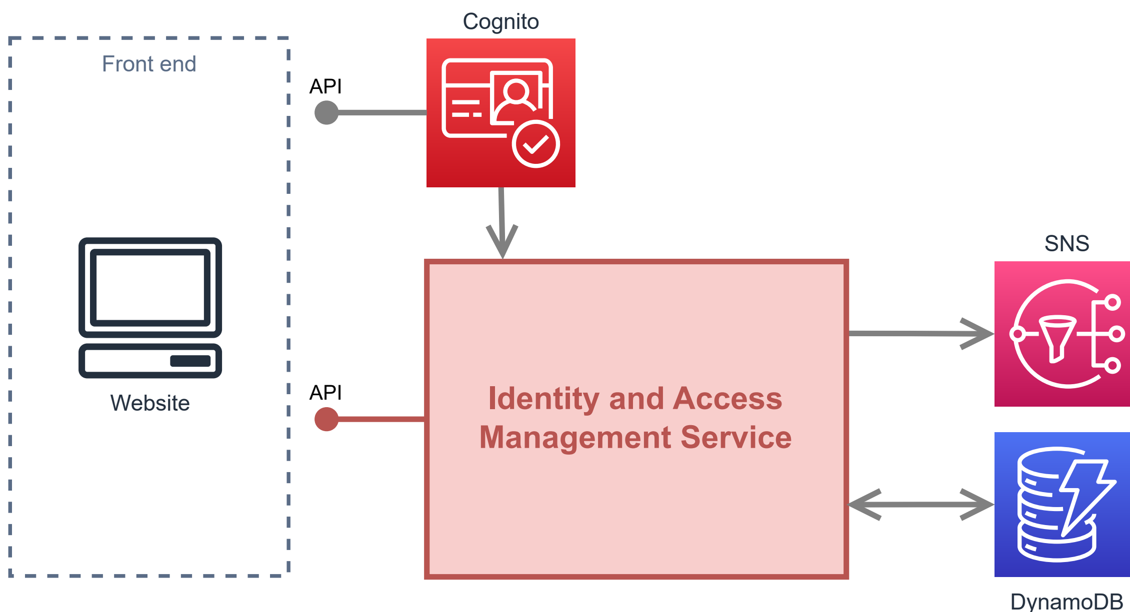
Logo, para a proposta deste trabalho, este serviço é simples, porém permite que suas funcionalidades sejam facilmente estendidas para realizar novas tarefas de gerenciamento conforme o produto avança e novas regras e políticas de acesso são necessárias.

Trata-se de um serviço composto por funções do AWS Lambda, o qual possui integração com o Amazon Cognito para permitir o cadastro e autenticação de novos usuários. Tarefas de gerenciamento são feitas a partir da API do serviço de IAM. Os dados do serviço são armazenados em um banco de dados do Amazon DynamoDB e a comunicação assíncrona com outros serviços ocorre por meio de tópicos do SNS. Assim, este serviço pode ser representado de forma simplificada conforme demonstrado na Figura 5.3.

5.4.2 Entidades e Banco de Dados

Para entender o serviço de IAM em mais detalhes, é preciso compreender as entidades presentes no serviço. Assim, nesta Seção, são apresentadas as entidades que compõem o serviço e como elas são representadas no banco de dados.

Figura 5.3: Visão geral do serviço de *Identity and Access Management* (IAM) do sistema de notificações.



5.4.2.1 Organization

Uma *Organization* representa o usuário do sistema de notificações, o qual está normalmente associado a uma empresa. Por simplicidade, neste trabalho, essa entidade foi representada apenas com dois campos: nome e e-mail. O e-mail é utilizado para identificar o usuário, visto que não é permitido existir dois usuários do sistema com o mesmo e-mail. Além disso, outras informações, como senha, são gerenciadas exclusivamente pelo Cognito.

5.4.2.2 Key

Uma *Key* representa uma chave de acesso à API. Essas chaves podem ser gerenciadas por *Organizations* para poderem acessar a API de envio de notificações mediante um método de autenticação baseado em *API keys*, o qual é comumente utilizado em APIs REST comerciais.

Esta entidade é representada através dos dados a seguir:

- ***id***: identificador único da chave de acesso gerado durante a criação;
- ***name***: nome descritivo da chave, utilizado para identificá-la de forma amigável;
- ***type***: identifica o tipo da chave. Chaves podem ser secretas (*secret*) ou publicáveis (*publishable*). As primeiras devem ser armazenadas em segurança, visto que são

utilizadas para permitir acesso à API. Já as chaves publicáveis podem ser utilizadas em aplicações *client-side*, como o *front-end*, e servem para identificar a aplicação;

- ***mode*:** identifica o modo como a chave é utilizada. As chaves podem possuir o modo *"live"*, caso sejam utilizadas em um ambiente de produção e o modo *"test"* caso sejam utilizadas em ambientes de teste;
- ***value*:** representa o valor da chave, o qual é uma *string* gerada aleatoriamente
- ***prefix*:** representa os primeiros 13 caracteres do valor da chave. O prefixo é utilizado para auxiliar a identificar chaves secretas após serem geradas.

5.4.2.3 Representação no Banco de Dados

Com base na descrição das entidades do serviço de IAM, o relacionamento entre elas pode ser representado conforme o diagrama da Figura 5.4. Através destes relacionamentos, os quais foram descritos nas Seções anteriores, modelaram-se as entidades no banco de dados DynamoDB. Os atributos de cada entidade foram abstraídos, visto que foram descritos nas Seções anteriores.

Figura 5.4: Diagrama de Entidade Relacionamento das entidades presentes no serviço de IAM do sistema de notificações.

5.4.3 Arquitetura e Funcionamento

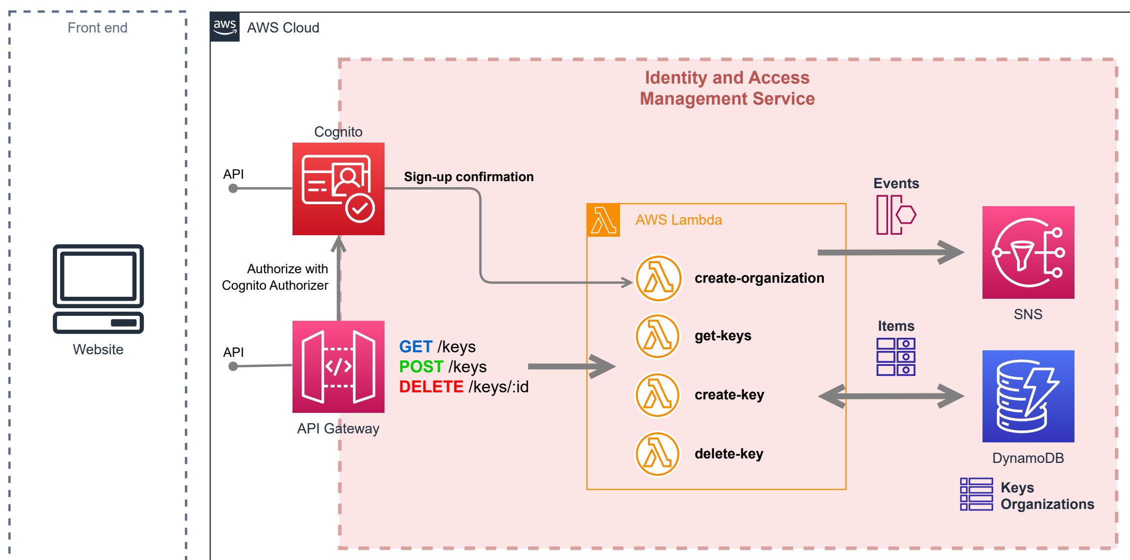
No diagrama da Figura 5.5 pode-se visualizar em detalhes a arquitetura interna do serviço de *Identity and Access Management*. Com base nesta Figura, descreveremos o funcionamento do serviço, bem como seus componentes.

Conforme comentado anteriormente, o serviço de IAM possui integração com o Amazon Cognito. Esta integração permite que os usuários do sistema de notificações interajam diretamente com a API do Cognito para realizarem o cadastro e *login* no sistema. A utilização do Cognito remove a necessidade de desenvolver uma solução própria e fornece uma solução de *front-end* básica para essas tarefas. Essa solução foi utilizada, sendo apresentada posteriormente. Com isso, ao construir a integração com o Cognito, no momento em que novos usuários realizarem o cadastro no sistema e confirmarem suas

contas através de um código enviado para o e-mail de cadastro, o Cognito inicializa a função *create-organization* do AWS Lambda, a qual configurara a conta do novo usuário no sistema de notificações.

Da mesma forma, a API do Cognito é utilizada para que usuários realizem a autenticação no sistema. Após a autenticação, usuários podem realizar requisições para as rotas expostas na API do serviço. Para gerenciar a API deste e dos demais serviços do sistema, utilizou-se o AWS API Gateway. Assim, toda requisição que chega ao serviço, é tratada inicialmente pelo API Gateway.

Figura 5.5: Arquitetura do serviço de *Identity and Access Management* (IAM) do sistema de notificações.



Dado que as rotas expostas por essa API necessitam que o usuário esteja autenticado, a autorização do usuário nestas rotas é realizada através da integração do Cognito e do API Gateway. Com isso, toda requisição é processada por uma função própria do Cognito chamada *Cognito Authorizer*. Esta função autoriza ou não a requisição. Caso seja autorizada, a requisição é encaminhada para a função do AWS Lambda adequada para tratá-la. No caso em que a requisição não é autorizada, o API Gateway interrompe o processamento e envia uma resposta de erro ao usuário.

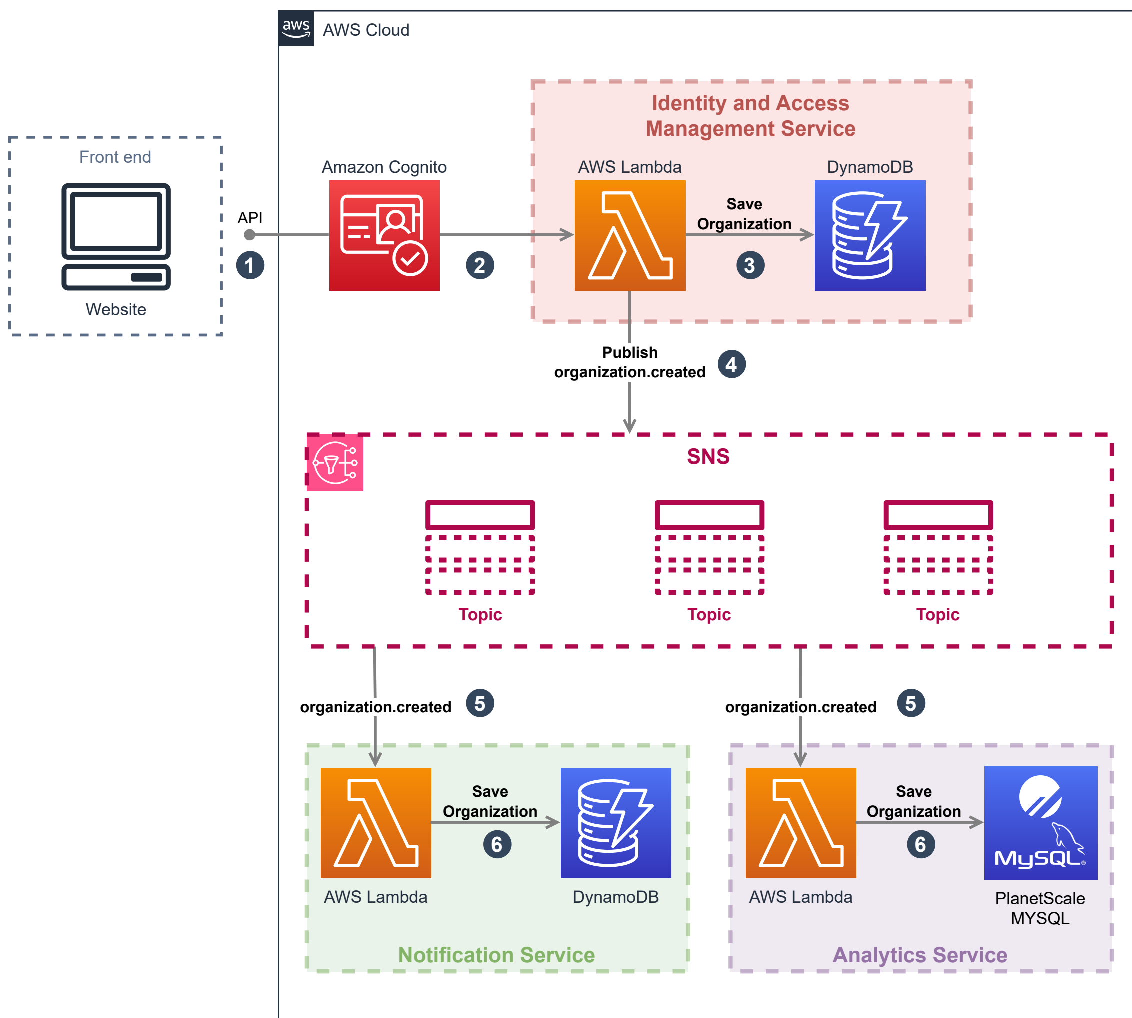
O API Gateway, em conjunto com o Amazon Cognito, são as portas que permitem acesso dos usuários ao serviço de IAM. O API Gateway é utilizado para direcionar as requisições recebidas pela API para a função do AWS Lambda que está configurada para tratar a requisição. Cada função do AWS Lambda costuma estar associada a uma rota da API. No diagrama da Figura 5.5 é possível verificar que para cada método da API, existe uma função lambda associada.

Ao observar o lado direito da Figura 5.5, percebe-se que o DynamoDB está presente no serviço, bem como o SNS. Este banco de dados é utilizado para armazenar as entidades presentes no serviço: *Organizations* e *Keys*. Já o SNS é utilizado para emitir os eventos que ocorrem no serviço de IAM.

Logo, o serviço de *Identity and Access Management* pode ser resumido como APIs expostas pelo API Gateway ou Cognito, as quais são conectadas às funções do AWS Lambda para processar requisições. Por fim, estas funções utilizam o DynamoDB para armazenamento de dados e SNS para mensageria.

5.4.3.1 Processo de sign-up

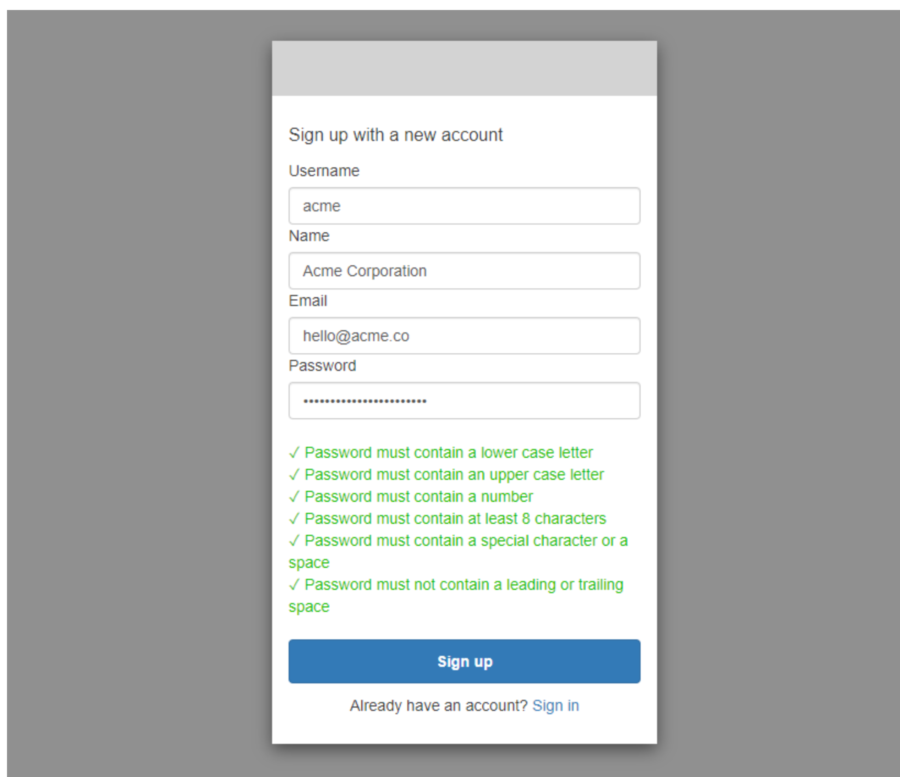
Figura 5.6: Processo de *sign-up* de novos usuários no serviço de *Identity and Access Management* (IAM) do sistema de notificações.



O processo de cadastro de novos usuários (*sign-up*), descrito na Figura 5.6, foi construído para utilizar a solução de *front-end* do próprio Amazon Cognito. Esta solução

fornece uma interface simples para realizar a criação de novas contas e também o acesso a elas. Com isso, não foi necessário implementar nenhuma interface própria ou utilizar a API do Cognito diretamente, o que facilitou e agilizou o desenvolvimento e os testes. Na Figura 5.7 pode ser visualizado a interface disponibilizada pelo Cognito.

Figura 5.7: Tela de *sign-up* do Cognito.



A etapa 1, identificada na Figura 5.6, corresponde ao processo de criação de conta conduzido através da interface do Cognito. Ao preencher os campos e clicar no botão de "Sign up", o Cognito envia um e-mail para o endereço contido no cadastro. Este e-mail contém um código de confirmação da conta. No momento em que a conta é confirmada, a etapa 2 ocorre, ou seja, o Amazon Cognito dispara um evento para a função `create-organization` do AWS Lambda.

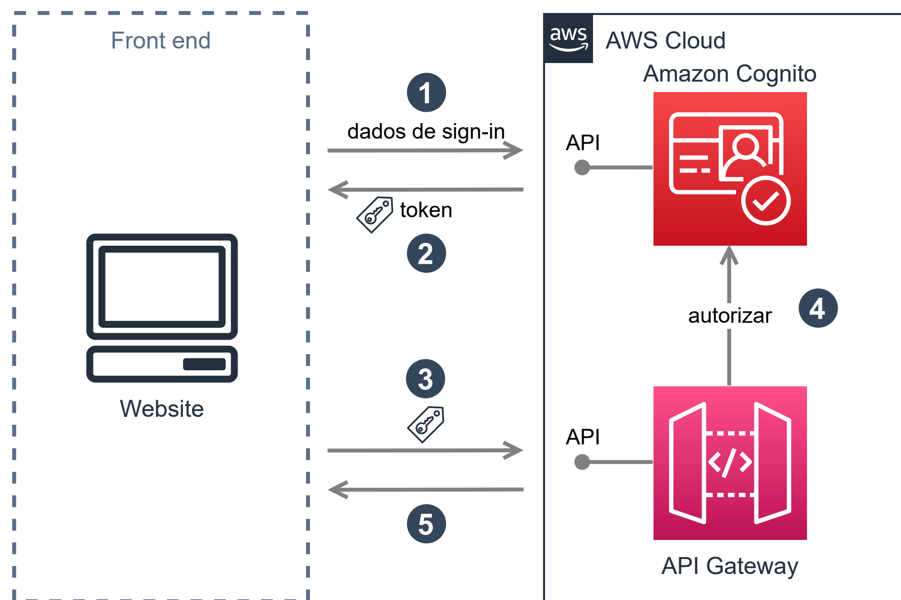
Esta função, ao ser invocada, realiza o registro do novo usuário na base de dados (etapa 3). Na sequência, na etapa 4, um evento de `organization.created` é emitido através do SNS no tópico `organization`.

Então, na etapa 5, os serviços de *Notification* e *Analytics*, os quais estão inscritos nesse tópico, recebem o evento. Por fim, o recebimento do evento invoca funções lambdas nos serviços, as quais realizam o registro do identificador do usuário na base de dados para que posteriormente possa ser utilizado para realizar o vínculo com outras informações.

5.4.3.2 Processo de autenticação e autorização

Após realizar o processo de *sign-up*, os usuários possuem contas válidas para acessar os serviços oferecidos pelo sistema de notificação. As rotas da API do sistema de notificações voltadas para o uso de uma aplicação *front-end* possuem um formato de autenticação diferente. O processo de autenticação e autorização para esse caso de uso pode ser visualizado em detalhes na Figura 5.8.

Figura 5.8: Processo de autorização para rotas voltadas para aplicações *front-end*.

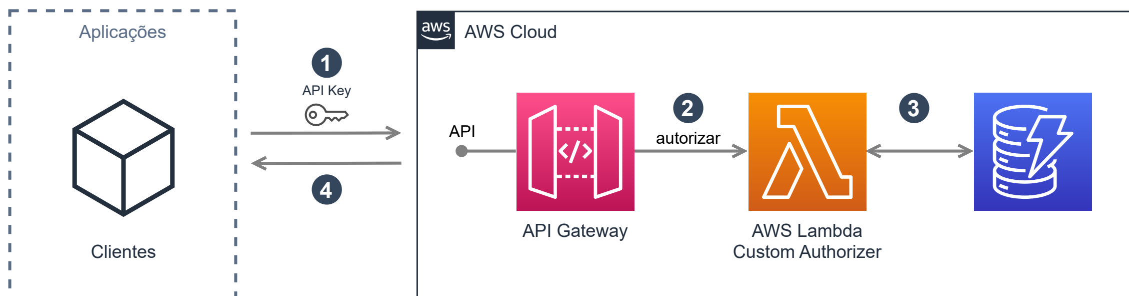


Primeiramente, a aplicação *front-end* envia uma requisição que é tratada pela API do Cognito. Esta requisição deve conter os dados de *sign-in*, como e-mail e senha. Em seguida, caso os dados estejam corretos, a API do Cognito responde com um *token* de acesso. Posteriormente, este *token* é utilizado nas requisições para autorizar o acesso à API dos serviços do sistema de notificação. Então, ao receber uma requisição, o API Gateway utiliza a integração com o Amazon Cognito para autorizar a requisição através do *Cognito Authorizer*. Por fim, o API Gateway retorna uma resposta contendo os dados do processamento da requisição ou um erro de não autorizado.

Para o caso de uso em que outros *softwares* desejam se integrar ao sistema de notificações para realizar envios, a autorização é feita de forma diferente. Neste caso, para realizar requisições é preciso possuir chaves de acesso (*Keys*), as quais são criadas através da API do serviço de IAM usando o método de autenticação descrito anteriormente. Após possuir uma chave de acesso, as requisições podem ser realizada para a API contendo a chave criada. Em seguida, o API Gateway utiliza a integração com o AWS Lambda para invocar uma função personalizada, a qual utiliza os dados contidos no banco de dados do

serviço de IAM para verificar se a chave recebida é válida ou não. Por fim, o API Gateway retorna a resposta adequada de acordo com o resultado da autorização. Este fluxo pode ser visualizado na Figura 5.9.

Figura 5.9: Processo de autorização para rotas voltadas para outros *softwares*.



5.4.4 Rotas da API

Nesta Seção, são descritas as rotas existentes na API do serviço de IAM, bem como o formato dos dados, o método de autenticação e autorização necessário e as respostas possíveis para cada uma das rotas.

5.4.4.1 Autenticação

Visto que as rotas presentes na API deste serviço são focadas em atividades de gerenciamento das contas e acesso das organizações, elas são voltadas para o uso de uma aplicação *front-end*. Por conta disso, o método de autenticação utilizado em todas as rotas desta API é o método descrito na Figura 5.8. A utilização deste método é baseada no envio de um *token* no cabeçalho da requisição HTTP. O valor fornecido neste campo deve conter o seguinte formato *Bearer <token>*, em que *token* refere-se ao valor recebido do Amazon Cognito durante o processo de autenticação.

5.4.4.2 Gerenciamento de chaves de acesso (API keys)

A API de envio de notificações autentica as requisições utilizando as *API Keys* registradas no ambiente do sistema de notificações. Assim, para que outras aplicações de *software* se integrem ao sistema de notificações, é necessário que elas utilizem estas chaves de acesso para autenticar suas requisições. Este método de autenticação está descrito em detalhes na Figura 5.9. Caso a chave fornecida seja inválida ou não exista,

erros são retornados nas requisições realizadas para API de envio indicando problemas de autenticação.

Para gerenciar as chaves de acesso vinculadas a uma *Organization*, utilizam-se as rotas presentes no serviço de IAM, as quais permitem que as chaves sejam consultadas, criadas e deletadas. Estas rotas estão descritas brevemente a seguir.

Consultar detalhes de uma chave de acesso: (`GET /keys/:id`) esta rota é utilizada para consultar, através do identificador único, os detalhes de uma chave de acesso previamente gerada. Em caso de sucesso, a rota responde com *status code 200 OK* e um objeto JSON contendo os detalhes da chave conforme os atributos da entidade *Key* descritos na Seção 5.4.2, exceto para casos de chaves com o tipo *secret*, em que o valor da chave não é retornado, pois só é visível no momento da criação. Por fim, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Gerar uma nova chave de acesso: (`POST /keys`) rota utilizada para gerar uma chave de acesso para ser utilizada para acessar a API de envio. É necessário fornecer um objeto JSON com os atributos *name*, *type* e *mode* de uma entidade de *Key*. Em caso de sucesso, a rota responde com *status code 201 Created* e um objeto JSON contendo os detalhes da chave gerada conforme os atributos da entidade *Key* descritos na Seção 5.4.2. Enfim, se houver algum erro de validação nos dados fornecidos, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

Deleta uma chave de acesso: (`DELETE /keys/:id`) rota utilizada para deletar, através do identificador único, os detalhes de uma chave de acesso previamente gerada. Em caso de sucesso, a rota responde apenas com *status code 200 OK*. Por fim, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Os detalhes completos sobre cada uma das rotas descritas nesta Seção podem ser encontrados na documentação da API presente no Anexo B, em conjunto com a descrição de cada campo e exemplos de requisições e respostas.

5.5 Módulo de *Notification Service*

Esta Seção visa descrever em detalhes o funcionamento do módulo de *Notification*. Para isso, é apresentada a visão geral do funcionamento do serviço, bem como os detalhes de sua arquitetura e API REST.

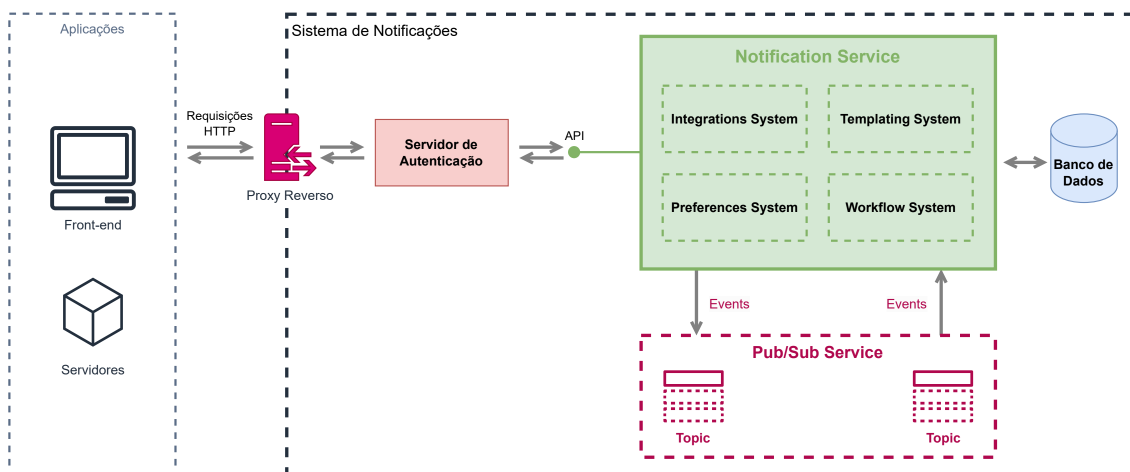
5.5.1 Visão Geral

O serviço de *Notification* é responsável por realizar todas as ações relacionadas ao envio de notificações e gerenciamento dos recursos relacionados ao envio. A definição de fluxos de envio de notificações, o gerenciamento de *templates*, a configuração de integrações com provedores e o gerenciamento de preferência e destinatários é através deste serviço. Logo, este é o principal serviço que compõe o sistema de notificações desenvolvido.

O objetivo deste serviço é prover as funcionalidades essenciais que um sistema de notificações deve possuir, conforme descritas na Seção 2.2. O sistema de *templating* implementado permite que *templates* possam ser construídos a partir de blocos de conteúdo, os quais podem ser reutilizados para que o mesmo *template* possua definições para múltiplos canais. Já em relação à lógica de roteamento das notificações, optou-se por implementar um construtor de fluxos de envio, o qual permite que os usuários definam suas necessidades de roteamento e jornadas de envio personalizadas. Por fim, para que destinatários possam definir suas preferências para o recebimento de notificações, implementou-se a possibilidade de usuários definirem conjuntos de preferências baseados em tópicos e assuntos. A partir destas definições as preferências dos destinatários podem ser controladas. O funcionamento desses e outros recursos são descritos em detalhes nas Seções seguintes.

Assim, podemos definir a arquitetura de alto nível da Figura 5.10 para representar os principais componentes do sistema de notificações necessários para o funcionamento do serviço. Nesta representação podemos visualizar a necessidade de um sistema de *proxy* reverso para rotear as requisições de aplicações para o serviço do sistema de notificações. Há também a necessidade de um servidor de autenticação para autorizar as requisições recebidas. Diversas tecnologias podem ser utilizadas para estes componentes, porém, neste serviço, optou-se por utilizar as funcionalidades API Gateway e suas integrações com autorizadores. Além disso, o serviço de *Notification* deve ser formado por uma série de módulos dedicados para cada funcionalidade e tais módulos podem ser implementados utilizando a tecnologia de computação desejada, porém neste trabalho optou-se por utilizar e um serviço de FaaS. Da mesma forma, sistemas de banco de dados devem ser usados para armazenar os dados pertinentes, sendo importante a utilização de um banco performático e escalável. Por fim, um sistema de mensageria ou de *pub/sub* distribuído pode ser usado para emitir e receber eventos atrelados ao processamento dos dados.

Figura 5.10: Arquitetura de alto nível do serviço de *Notification* do sistema de notificações.



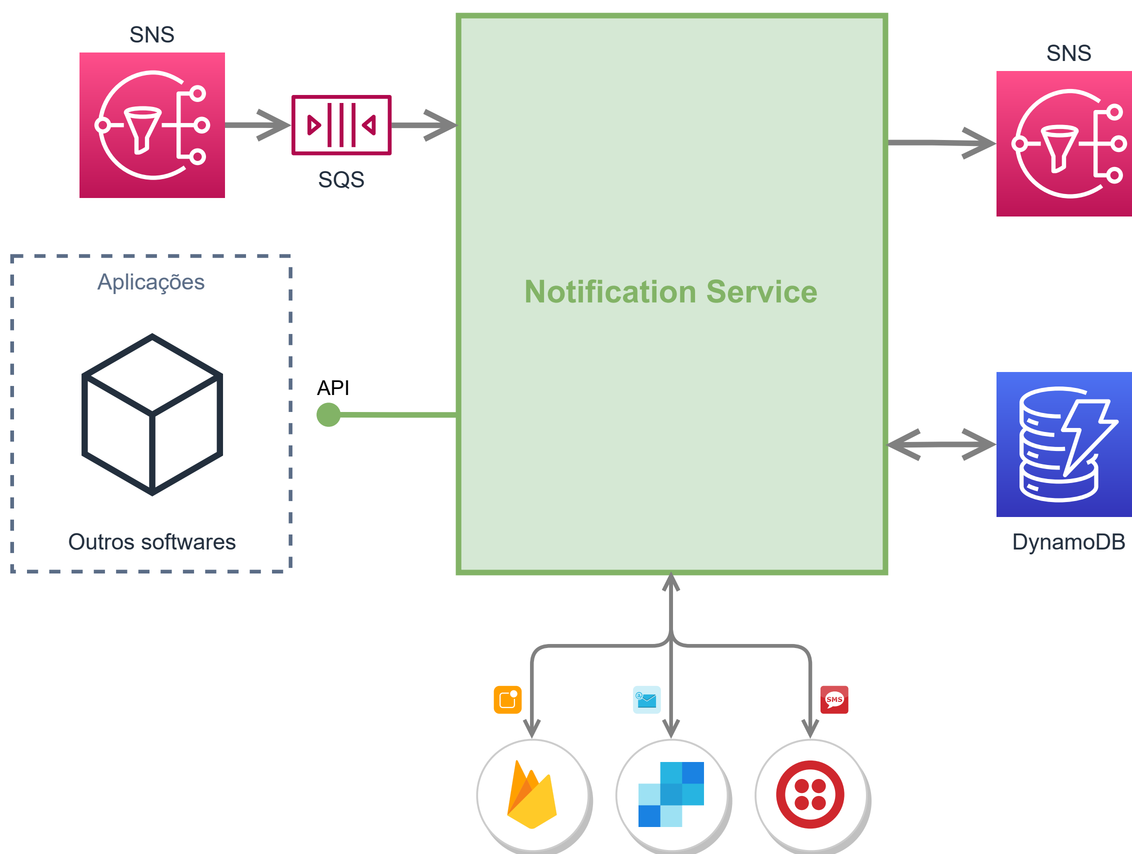
Consequentemente, ao basear-se na arquitetura de alto nível para os objetivos citados serem alcançados, optou-se por desenvolver o *Notification Service* como um conjunto de funções do AWS Lambda que se comunicam entre si de forma assíncrona através de eventos para cooperar na tarefa de enviar notificações. Além disso, foram usadas funções lambda para implementar a API REST que permite o envio de notificações e o gerenciamento do ambiente. Para obter um desempenho consistente em relação às escritas e leituras no banco de dados, o DynamoDB foi utilizado como banco principal. Ao utilizar o DynamoDB, o *overhead* adicionado pelo estabelecimento de conexões com o banco é reduzido, o que garante maior agilidade no envio de notificações.

Assim, de maneira geral, o serviço de *Notification* pode ser representado como na Figura 5.11: um módulo composto por funções do AWS Lambda, as quais utilizam tópicos do SNS e filas do SQS para comunicação assíncrona e que também expõe uma API REST através da utilização do API Gateway integrado às funções lambda.

5.5.2 Entidades e Banco de Dados

Para entender o serviço de *Notification* em mais detalhes, é preciso compreender as entidades presentes no serviço. Assim, nesta Seção, são apresentadas as entidades que compõem o serviço e como elas são representadas no banco de dados.

Figura 5.11: Visão geral do serviço de *Notification* do sistema de notificações.



5.5.2.1 *Organization*

Assim, como no serviço de IAM, uma *Organization* representa o usuário do sistema de notificações. Conforme visto anteriormente, os dados de *Organization* são replicados para o serviço de *Notification*. Logo, essa entidade é composta por um identificador da *Organization*, normalmente, e-mail e, opcionalmente, o nome da organização.

5.5.2.2 *Integration*

Usuários que desejem enviar notificações precisam configurar a integração com os provedores e canais desejados para que o sistema de notificações possua os dados necessários para poder enviar notificações. A configuração destas integrações é feita por meio de entidades de *Integration*. Para cada provedor utilizado por um usuário, uma instância da entidade de *Integration* deve ser criada. Cada provedor exige dados de configuração diferentes conforme as especificações da API própria do provedor. Através da utilização do sistema de notificações, basta que estes dados sejam fornecidos para que notificações sejam enviadas, ou seja, não é necessário desenvolver nenhum tipo de integração direta

com o provedor.

Neste trabalho, foi desenvolvido suporte a três canais de distribuições: e-mail, SMS e *push notifications*. Para cada um dos canais de distribuição, foi implementado integração com um provedor. Para o canal de e-mail, integramos com o provedor *SendGrid*. Já para o canal de SMS, integramos com o provedor *Twilio*. Por fim, o envio de *push notifications* para dispositivos Android e iOS foi realizado através da integração com o provedor *Firebase Cloud Messaging* (FCM).

5.5.2.3 Element

Um *Element* é um bloco de conteúdo utilizado para compor mensagens enviadas por meio de notificações. Através dos diferentes tipos de *elements* disponíveis, as mensagens podem ser compostas de forma simples, sem a necessidade de conhecer alguma forma complexa de representação de *templates*. A representação de mensagens através da utilização destes blocos, permite a construção de uma interface de usuário, a qual possibilita que usuários sem conhecimento técnico criem *templates* para notificações com mecanismos de "arrastar e soltar" através da composição destes *building blocks* chamados de *elements*. Eles servem como uma abstração para facilitar a construção e representação de *templates*.

Inicialmente, foram implementados quatro tipos de *elements*:

- ***text*:** este elemento é utilizado para adicionar textos ao conteúdo da notificação. Estes textos podem ser incluídos no formato de título 1, título 2 e parágrafos, bem como conter estilizações, por exemplo, negrito, itálico e sublinhado;
- ***action*:** com elementos deste tipo é possível adicionar *links* para páginas da Internet;
- ***image*:** utilizado para adicionar imagens ao *template* e conteúdo da notificação. É possível personalizar o elemento ao adicionar *link*, descrição e largura da imagem;
- ***meta*:** este elemento é utilizado para adicionar metadados ao *template* da notificação, por exemplo, um título, o qual pode ser utilizado nos canais que possuem tais informações, como o assunto de um e-mail e o título de uma notificação *push*.

No momento em que as notificações são enviadas, os elementos que compõem o conteúdo delas são convertidos do formato de coleções de elementos para uma representação adequada para o canal e provedor sendo utilizado. Existe a possibilidade de que nem todo canal possua suporte a todos *elements*. Nestes casos, podem ser implementadas alternativas de representação ou, então, esses elementos podem ser completamente igno-

rados. Por exemplo, imagens podem ser representadas em mensagens de SMS como um *link* para a imagem.

Além disso, o conteúdo presente em cada um dos tipos de elemento pode ser personalizado das variáveis disponíveis no ambiente no momento do envio da notificação, por exemplo, os dados do destinatário. Desta forma, os usuários podem especificar conteúdos dinâmicos para suas notificações. Para acessar variáveis do ambiente de envio, pode-se referenciar o nome da variável dentro de `{{` e `}}`, por exemplo, `{{recipient.name}}` para acessar o nome do destinatário.

Por fim, foi implementado um conjunto básico de tipos de elementos, porém este conjunto pode ser facilmente estendido para que o sistema de notificações suporte novos tipos. Já em relação ao banco de dados e as entidades *Elements* é importante ressaltar, que estas não são encontradas no Diagrama de Entidade Relacionamento, pois foram representadas como atributos de outras entidades.

5.5.2.4 Template

No sistema de notificações, um *template* é um modelo de notificação que pode ser reutilizado diversas vezes para enviar notificações para múltiplos destinatários. Notificações para desejar felicidades no aniversário de destinatários e notificações para informar acontecimentos em redes sociais são exemplos comuns de notificações que podem se beneficiar de um *template*.

Um *template* é construído através de um conjunto de *elements* e pode possuir definições especificas para cada canal suportado pelo sistema. Assim, um mesmo *template* pode possuir um modelo de estruturação de conteúdo voltado para SMS, *push notifications* e e-mail. Através desse recurso, o sistema de roteamento de notificações pode utilizar um mesmo *template* para enviar notificações para diferentes canais.

Além disso, um *template* possui outros atributos importante, por exemplo, o vínculo com tópicos de preferência. Este e outros atributos são descritos a seguir:

- ***id*:** identificador único do *template*;
- ***title*:** atributo que indica o título ou nome do *template*, usado para facilmente identificar esta entidade;
- ***topicId*:** opcionalmente, um *template* pode ser vinculado a um tópico de preferência, o que faz com que as notificações usando o *template* só sejam enviadas para o destinatário caso ele tenha optado por recebê-las;

- ***layouts*:** neste atributo são armazenados os diferentes modelos de conteúdo do *template* específicos para cada canal no formato de um objeto chave-valor, em que as chaves são o canal e o valor o modelo de conteúdo especificado usando *elements*.

5.5.2.5 Recipient

A entidade de *Recipient* representa os destinatários das notificações, os quais são registrados no sistema de notificação e vinculados a uma organização. Um *recipient* é identificado por um ID, o qual é definido no momento da criação de um novo *recipient* no sistema de notificações. Este identificador deve ser um valor que permita aos *softwares* integrados ao sistema de notificações reconhecer o *recipient*. Assim, recomenda-se que o identificador utilizado seja o mesmo valor usado para identificar o *recipient* no *software* de origem.

Além disso, esta entidade pode conter outros dois tipos de informação importantes durante o processo de envio: dados vinculados a canais de distribuição e dados do perfil do *recipient*. O primeiro é usado para permitir o envio através de um canal, por exemplo, o número de celular para permitir o envio de SMS. Já o segundo são informações que podem ser usadas para preencher dados dinâmicos nas notificações. A seguir, os atributos desta entidade são descritos em detalhes:

- ***id*:** identificador único do *recipient*, recomenda-se a utilização do identificador do *recipient* no *software* de origem deste destinatário;
- ***email*:** endereço de e-mail do *recipient*, obrigatório para permitir o envio de e-mails para o destinatário;
- ***phoneNumber*:** número de telefone do *recipient*, obrigatório para permitir o envio de SMS para o destinatário;
- ***firebaseToken*:** é o *token* do dispositivo *mobile* (Android ou iOS) do *recipient*, o qual garante que o destinatário deu permissão para que o *software* envie notificações *push* para ele. Este campo é obrigatório para que *push notifications* sejam enviadas;
- ***profile*:** atributo representado como um objeto chave-valor, o qual contém informações adicionais sobre o destinatário que podem ser utilizadas para compor as notificações enviadas a ele, por exemplo, nome e data de nascimento.

5.5.2.6 *Preference Section*

Uma *Preference Section* é utilizada para agrupar preferências de destinatários que estejam relacionadas de alguma forma. Assim, uma seção de preferências é composta por um conjunto de tópicos e atributos que permitam a identificação e configuração da entidade. A seguir estão descritos os atributos de uma *Preference Section*:

- ***id*:** identificador único da *Preference Section*;
- ***name*:** nome da seção, utilizado na exibição das preferências para os destinatários;
- ***channelOptions*:** lista de canais de distribuição que os tópicos vinculados a esta seção poderão realizar o envio de notificações. Os destinatários poderão gerenciar suas preferências de modo que escolham quais canais são permitidos para cada tópico pertencente à seção.

5.5.2.7 *Preference Topic*

Um *Preference Topic* é utilizado para agrupar *templates* de notificações que estejam relacionados a um mesmo assunto. Assim, um destinatário pode gerenciar suas preferências através da escolha de optar ou não por receber notificações de determinados assuntos. A organização de preferências de destinatários em seções e tópicos fornece grande poder ao sistema de notificações e aos destinatários, visto que permite uma alta granularidade no controle de preferências. Por fim, um tópico de preferência é composto por *templates* e configurações. Os atributos de um *Preference Topic* são descritos a seguir:

- ***id*:** identificador único da *Preference Topic*;
- ***name*:** nome do tópico, utilizado na exibição das preferências para os destinatários identificarem o assunto das notificações que podem escolher receber;
- ***linkedTemplatedIds*:** define a lista de *templates* que estão vinculados ao assunto do tópico de preferência;
- ***defaultState*:** este atributo define o comportamento padrão do tópico quando o destinatário ainda não definiu sua preferência e pode conter um dos seguintes valores: `opted_in`, `opted_out` e `required`. Quando o comportamento padrão é enviar a notificação, atribui-se o valor `opted_in`. Já quando o comportamento padrão é não enviar a notificação, o valor `opted_out` é utilizado. O valor `required` funciona como o `opted_in`, porém não permite que o usuário opte por não receber a notificação, ou seja, pode ser usado quando o tópico está associado a um assunto

crítico, o qual não pode ser perdido.

Ao estruturar as preferências dos destinatários desta forma, sempre que uma notificação for enviada através de um *template*, as configurações de preferência do destinatário são verificadas para garantir de que ele realmente deve receber a notificação em questão. Para exemplificar a estruturação de preferências de destinatários, podemos imaginar uma seção de preferências chamada "Cobranças e Faturamentos", a qual é destinada para agrupar tópicos relacionados a este assunto. Esta seção poderia ser composta por dois tópicos, por exemplo, "Alertas de custo" e "Lembretes de pagamento". Cada um destes tópicos pode estar vinculado a múltiplos *templates* que definem modelos de conteúdo relacionados a estes assuntos.

5.5.2.8 Workflow

Para permitir a customização dos fluxos de envio e lógicas de roteamento, *Workflows* podem ser criados ao combinar funções especiais, canais de distribuição e condições lógicas. Assim, os *Workflows* são compostos por uma sequência de etapas. Cada etapa pode realizar o envio de uma notificação através de um canal ou executar uma função especial. Além disso, para cada etapa, condições lógicas podem ser atribuídas, as quais são avaliadas antes da execução da etapa para decidir se ela deve ser executada ou não.

Estas entidades foram construídas de maneira que interfaces de usuário possam ser desenvolvidas para permitir que usuários criem seus próprios *Workflows* através de uma estratégia simples de "arrastar e soltar" etapas. Neste trabalho, implementou-se dois tipos de etapas:

- **Etapa de envio de notificação (*send*):** ao executar essa etapa de um *Workflow*, uma notificação é enviada para um destinatário através do canal selecionado na etapa, por exemplo, SMS, e-mail ou *push notification*. Além disso, um *template* é vinculado a cada etapa de envio para definir o conteúdo da notificação. O envio só ocorre se todas as condições vinculadas à etapa são satisfeitas;
- **Etapa de espera (*delay*):** este tipo de etapa é utilizado para que a execução de um *Workflow* seja interrompida por um determinado intervalo de tempo, por exemplo, aguardar uma hora para executar a próxima etapa. Assim, como nos demais tipos de *Workflow*, todas as condições devem ser satisfeitas para a etapa ser executada.

As condições vinculadas a uma etapa possuem três componentes:

- **propriedade:** este componente identifica qual propriedade das variáveis disponíveis no ambiente de envio é utilizada para avaliar a condição;
- **valor:** este componente define o valor utilizado para comparar ao valor contido na variável especificada na componente "propriedade";
- **operação:** este componente determina qual comparação é utilizada para avaliar a propriedade e o valor da condição. Neste trabalho, foram implementadas as operações de: "maior que" (>), "menor que" (<), "maior ou igual que" (>=), "menor ou igual que" (<=), "igual a" (==), "diferente de" (`!=`), "vazio", "não vazio".

Caso fosse desejado adicionar uma condição a uma etapa para que ela só seja executada se o destinatário more em Porto Alegre, a componente "propriedade" deveria conter o valor `{{recipient.profile.city}}` para referenciar a variável `city` do perfil de um destinatário. A componente "valor", obviamente, deveria conter a *string* `"Porto Alegre"` e a operação selecionada deveria ser "igual a" (==).

Por fim, o fluxo de etapas definidas por um *Workflow* é executado durante o processamento de um *Workload*. Conforme veremos a seguir, para cada destinatário presente em um *Workload*, o *Workflow* é executado uma vez.

5.5.2.9 Workload

Um *Workload*, como o nome indica, refere-se a uma carga de trabalho no sistema de notificações. Neste contexto, uma carga de trabalho é a tarefa de envio de notificações para um conjunto de destinatários. Uma carga de trabalho pode ser inicializada para realizar o envio de notificações de quatro formas distintas:

- **Envio simples (`simple`):** a carga de trabalho realiza o envio de apenas uma notificação para cada destinatário contido no *Workload* e o conteúdo da notificação é especificado através de dois campos: `title` e `body`. Ambos os campos devem conter *strings* com conteúdo utilizado na construção da notificação. Este tipo, é basicamente uma abstração de *elements*, pois internamente estes campos são convertidos para uma estrutura baseada em blocos de elementos;
- **Envio com *template in-line* (`element`):** neste tipo de carga de trabalho, apenas uma notificação é enviada por destinatário e o conteúdo da notificação é definido através de um conjunto de *elements*, da mesma forma que um *template* é representado;
- **Envio através de *templates* (`template`):** apenas uma notificação é enviada por

destinatário e o identificador único de um *template* é fornecido para definir o conteúdo da notificação;

- **Execução de um *workflow*** (`workflow`)**:** neste tipo de carga de trabalho, o identificador único de um *Workflow* é utilizado para definir o fluxo de envio executado para cada destinatário. Por se tratar da execução de um *Workflow*, existe a possibilidade de que um mesmo destinatário receba múltiplas notificações caso existam múltiplas etapas de envio de notificações.

Durante o processamento dos *Workloads* que não estão vinculados a um *Workflow* definido pelo usuário, ou seja, para os *Workloads* do tipo `simple`, `element` e `template`, um *Workflow* é definido em tempo de execução. Estes *Workflows* construídos em tempo de execução possuem apenas uma etapa de envio e nenhuma condição aplicada. Além disso, dado que tópicos de preferências de destinatários são vinculadas a *templates*, no caso de cargas de trabalho do tipo `simple` e `element`, as preferências dos destinatários não são avaliadas e o envio sempre ocorre.

Assim, um *Workload* é composto por uma lista de destinatários, para os quais deseja-se enviar uma notificação ou executar um *Workflow*, e o tipo de envio de notificação que pode ser um dos descritos anteriormente.

5.5.2.10 Workload Run

Conforme visto anteriormente, um *Workload* contém uma lista de destinatários que é utilizada para realizar o envio de notificações. Entretanto, um *Workflow*, criado por usuários ou em tempo de execução, é executado para um destinatário específico. Assim, há a necessidade de segregar a carga de trabalho definida em uma entidade de *Workload* em cargas de trabalho menores para cada destinatário. Durante o processamento de *Workloads* esta tarefa é realizada e dá origem a múltiplas entidades de *Workload Runs*, as quais representam a carga de trabalho associada a execução de um *Workflow* para um único destinatário.

Logo, um *Workload Run* é composto por três atributos:

- `recipientId`: identificador único do destinatário, para o qual um *Workflow* é executado através desta carga de trabalho;
- `workloadId`: identificador único do *Workload* que deu origem a esta carga de trabalho volta para um destinatário específico;
- `currentWorkflowStep`: número que indica a etapa atual do *Workflow* sendo exe-

cutado por esta carga de trabalho.

5.5.2.11 Notification

Durante a execução de um *Workflow*, seja ele criado previamente por um usuário ou em tempo de execução, notificações podem ser enviadas por meio de canais específicos. Estas notificações são representadas por uma entidade de *Notification*, a qual possui os seguintes atributos:

- `id`: identificador único da notificação;
- `recipientId`: identificador único do destinatário, para o qual a notificação é enviada;
- `workloadId`: identificador único do *Workload* que deu origem a esta notificação;
- `status`: o status atual da notificação, este atributo é descrito em maiores detalhes na sequência;
- `channel`: o canal em que a notificação é enviada, no momento há suporte para os seguintes valores: `email`, `sms` e `push`;
- `message`: o conteúdo que é enviado na notificação, o qual pode ser representado como o identificador único de um *template* ou uma estrutura de *elements*;

Durante o processamento da notificação, mudanças de status podem ocorrer e cada um deles representa um momento no ciclo de vida de uma notificação. Os possíveis valores de status estão descritos a seguir:

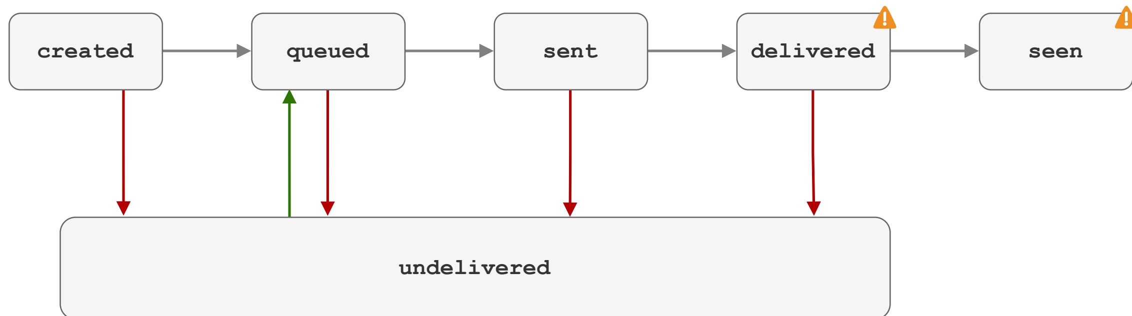
- `created`: representa o estado da notificação logo após sua criação durante a execução de uma etapa de envio em um *Workflow*;
- `queued`: estado em que a notificação está enfileirada aguardando ser consumida por uma das funções do AWS Lambda para ser processada;
- `sent`: estado no qual o sistema de notificações realizou a comunicação com o provedor de envio e esta ocorreu com sucesso, o que garante que a notificação foi enviada ao provedor para ser processada;
- `delivered`: representa o estado da notificação ao ser recebida pelo destinatário, a presença deste estado no ciclo de vida depende do canal em que a notificação foi enviada e do provedor, visto que nem todos canais e provedores possuem suporte a este status;
- `undelivered`: estado em que houve algum problema durante o processamento da

notificação e não foi possível enviá-la. Às vezes o problema não é definitivo, o que faz com que a notificação possa voltar a ser enfileirada para processamento;

- `seen`: indica que a notificação foi lida pelo destinatário. No caso de um e-mail, por exemplo, significa que o destinatário abriu o e-mail. Novamente, nem todos canais e provedores dão suporte a este status.

Assim, pode-se representar o ciclo de vida de uma notificação conforme o diagrama da Figura 5.12. Nesta Figura é possível notar claramente que há a possibilidade de uma notificação transicionar do status de *undelivered* para *queued*. Os status que podem não ocorrer devido a limitações do canal ou provedor estão destacados na Figura.

Figura 5.12: Transições de *status* presentes no ciclo de vida de uma notificação.



5.5.2.12 *Notification Event*

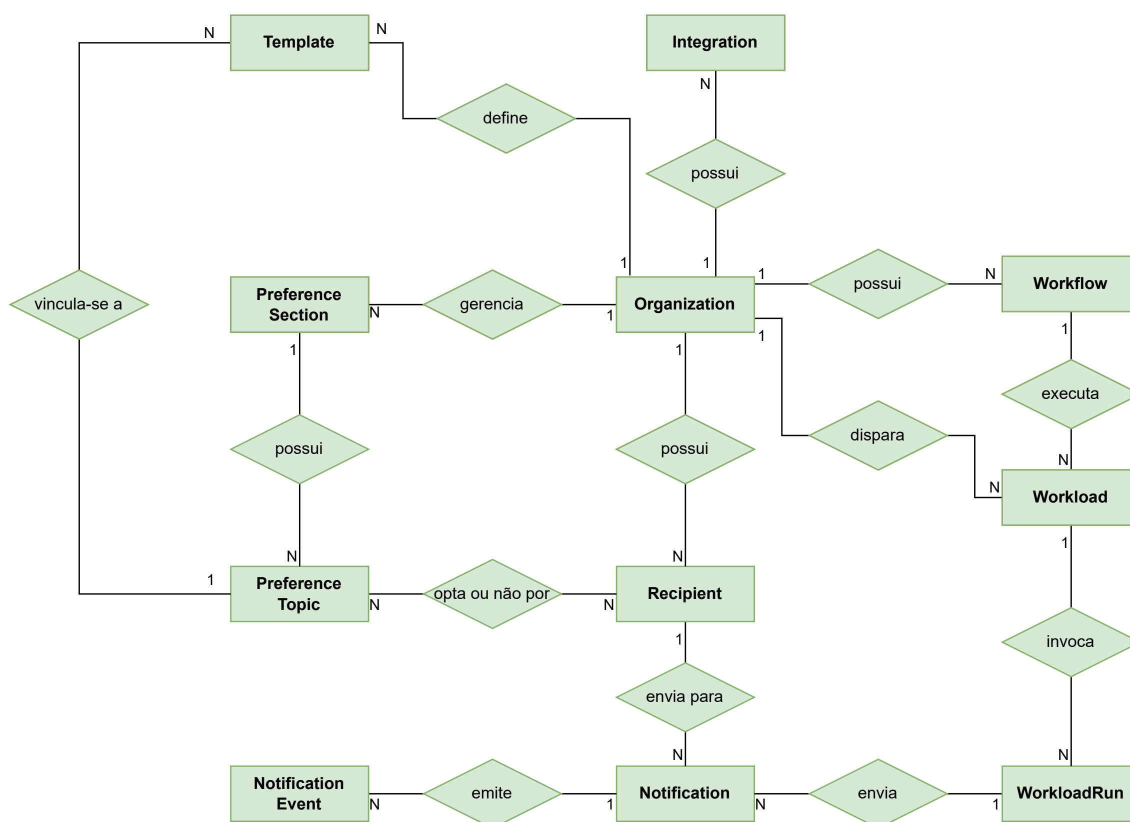
As entidades de *Notification Event* representam os eventos atrelados a uma notificação. Cada evento representa um fato ocorrido com a notificação e tais fatos podem ser os motivadores para causar transições de status na notificação. No geral, todo status de uma notificação pode ser associado a um evento que ocasionou a ocorrência do status. Entretanto, nem todo evento necessariamente implicam em uma alteração de status.

Para o contexto deste trabalho, os tipos de eventos disponíveis são os mesmos valores disponíveis para o status de uma notificação. Todo evento armazena a data e hora em que ele ocorreu, bem como um campo opcional que pode conter dados em formato de JSON com contexto atrelado ao evento. Por exemplo, no caso do evento de `sent`, este campo contém a resposta recebida do provedor ao realizar a requisição de envio e, no caso de um evento `undelivered`, o campo pode conter detalhes sobre o problema ocorrido.

5.5.2.13 Representação no Banco de Dados

Com base nas descrições dadas nas Seções anteriores para cada uma das entidades e nos requisitos do serviço de *Notification*, definiu-se o relacionamento entre as entidades conforme o diagrama de entidade relacionamento da Figura 5.13. As entidades foram representadas e modeladas no banco de dados DynamoDB através destes relacionamentos. Os atributos presentes em cada entidade não foram representados no diagrama para simplificar a visualização, visto que já foram citados e descritos nas suas respectivas Seções.

Figura 5.13: Diagrama de Entidade Relacionamento das entidades presentes no serviço de *Notification* do sistema de notificações.

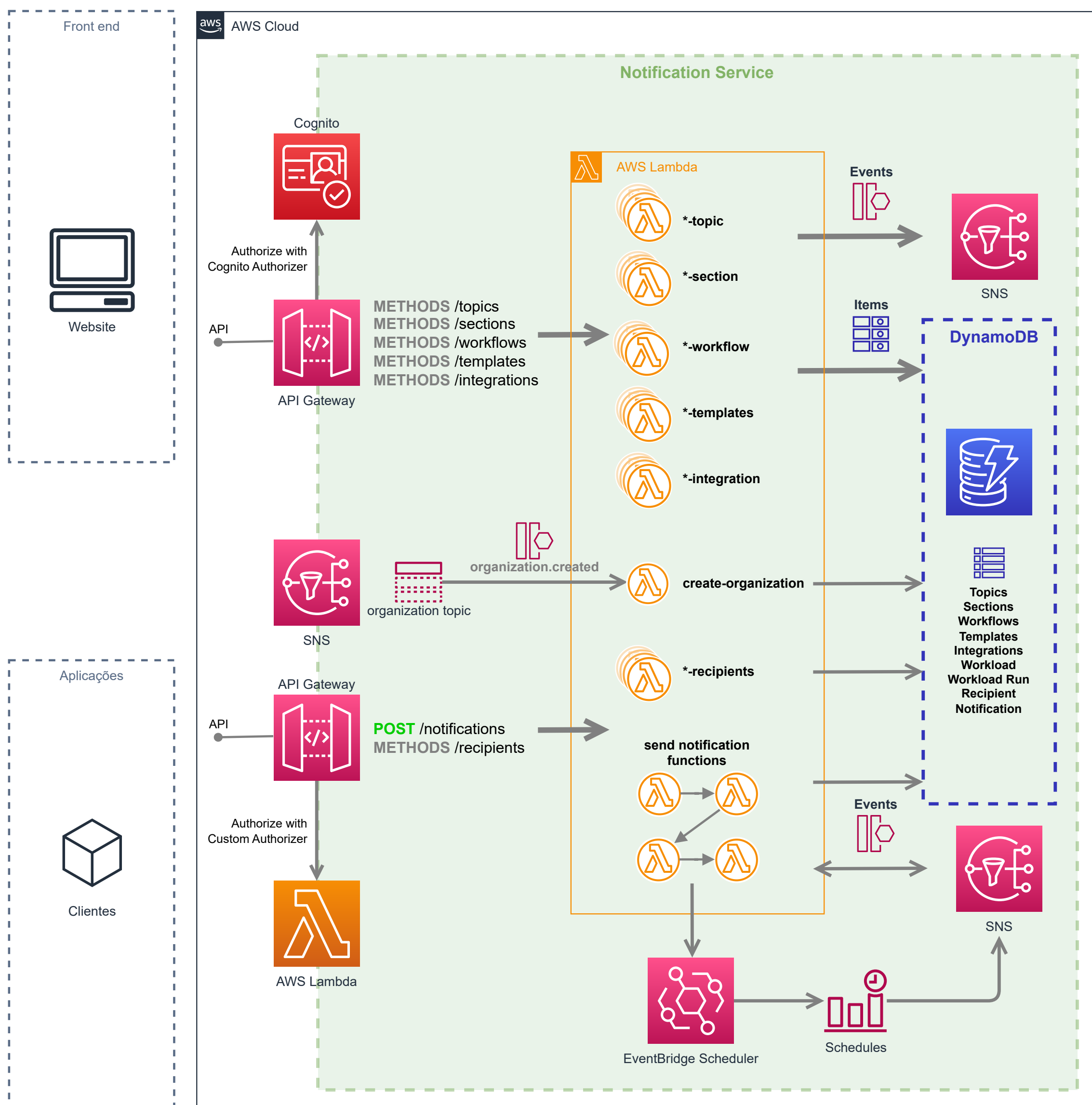


5.5.3 Arquitetura

No diagrama da Figura 5.14, pode-se visualizar em maiores detalhes a arquitetura interna do serviço de *Notification*. Com base nesta Figura, descrevemos o funcionamento do serviço, bem como seus componentes. Alguns detalhes dos componentes internos foram omitidos desta imagem para facilitar a visualização. Entretanto, durante a descrição dos componentes, novos diagramas são adicionados, os quais contêm o detalhamento

completo.

Figura 5.14: Detalhes da arquitetura do serviço de *Notification* do sistema de notificações.



De maneira geral, na Figura 5.14 está representada a API REST do serviço de *Notification*, a qual possui rotas voltadas para aplicações *front-end* e rotas voltadas para que outras aplicações de *software* se integrem ao sistema de notificações. Nesta representação, foram identificadas os principais recursos da API REST do serviço. Para facilitar a visualização, os recursos que possuem múltiplos métodos foram representados com o prefixo "METHODS"ao invés da especificação de cada um dos métodos HTTP. Posteriormente, os métodos de cada recurso são descritos em mais detalhes na Seção 5.5.4.

Ainda em relação à API REST do serviço, podemos identificar os métodos de autenticação utilizados em cada rota. As rotas destinadas para aplicações *front-end* são autenticadas através do *Cognito Authorizer* com o uso de *tokens*. Já as rotas voltadas

para outras aplicações de *software* são autenticadas através de um *Custom Authorizer* implementado utilizando uma função do AWS Lambda, a qual valida as chaves de acesso da API recebidas na requisição.

Assim, como nos demais serviços do sistema de notificação, a API REST é exposta através do serviço Amazon API Gateway. Os recursos e métodos definidos no API Gateway estão integrados a funções do AWS Lambda, as quais são executadas ao receber uma requisição. Para cada rota existente, uma função específica foi construída para tratar as requisições. No serviço, essas funções foram comumente identificadas com nome, como `create-workflow`, `delete-workflow` e `upsert-template`, ou seja, foram utilizados prefixos para identificar a operação implementada pela função para uma determinada entidade. Por isso, no diagrama da Figura 5.14 existem grupos de funções nomeadas com o prefixo "*"para representar que existem múltiplas operações relacionadas à entidade.

Através das rotas planejadas para serem utilizadas por uma aplicação *front-end*, é possível realizar o gerenciamento de várias entidades importantes para o serviço de *Notification*. Entende-se que uma aplicação *front-end* será desenvolvida posteriormente para que integrações com provedores sejam configuradas, *templates* sejam criados, preferências de usuários sejam definidas através de seções e tópicos, bem como *Workflows* sejam construídos para definir os fluxos de envio personalizados do usuário. Logo, essa API permite que o ambiente do sistema de notificações de um usuário seja configurado para enviar notificações aos destinatários utilizando as entidades configuradas sempre que requisições de envio são recebidas.

Toda configuração do ambiente do sistema de notificação de um usuário é armazenada em tabelas do DynamoDB. Além disso, sempre que modificações são realizadas através da API no ambiente de um usuário, eventos são emitidos para notificar outras funções ou outros serviços do sistema de notificações interessados nestes acontecimentos. A distribuição destes eventos ocorre através do DynamoDB Streams e do SNS por meio de um padrão chamado *Transactional Outbox*, o qual está descrito em detalhes na Seção 5.5.3.1. No diagrama da Figura 5.14 optou-se por omitir os detalhes do padrão para priorizar a simplicidade, porém ele está representado em outros diagramas.

Conforme visto na Figura 5.6, quando novos usuários se cadastram no sistema de notificações, o serviço de IAM emite um evento para notificar este acontecimento. O serviço *Notification* funciona como um *subscriber* para estes eventos de `organization.created`, visto que para permitir que usuários configurem seu ambiente no sistema de

notificações, é necessário que ele tenha conhecimento da existência dos usuários. Assim, no centro da Figura 5.14 podemos identificar a função do AWS Lambda que é invocada neste processo de replicação de dados. A função `create-organization` é responsável por realizar o registro no serviço de *Notification*, o que permite o uso do ambiente de envio.

Já ao avaliarmos a parte inferior da Figura 5.14, onde está representada a API REST utilizada por *software* construído pelos usuários, podemos identificar como a integração com o sistema de notificações é realizada.

Primeiramente, para que o envio de notificações possa ocorrer, é necessário que destinatários estejam registrados no sistema de notificações. Para realizar esta tarefa, existem rotas que permitem o gerenciamento dos destinatários por parte do *software* integrado ao sistema de notificações. Assim, através delas é possível sincronizar os dados de um destinatário entre o sistema de notificações e o *software* integrado a ele. Dado que as informações dos destinatários são propriedade de outro *software*, optou-se por manter a sincronização destas informações como uma responsabilidade da comunicação entre o *software* integrado e o sistema de notificação. Esta decisão está relacionada ao fato de que os destinatários para o sistema de notificação são, na realidade, os usuários do *software* integrado. Assim, quando um novo usuário passar a existir no *software* do cliente do sistema de notificação, ele pode ser registrado programaticamente como um destinatário no sistema de notificações ao enviar requisições para as rotas adequadas. Na Figura 5.14, podemos visualizar que estas rotas são tratadas por funções do AWS Lambda representadas como `*-recipient`.

Também na API de integração com o sistema, temos a rota de envio de notificações, a qual é responsável por invocar as funções que dão início ao fluxo de envio de notificações. No diagrama da Figura 5.14, as funções do AWS Lambda responsáveis pelo fluxo de envio estão representadas de forma simplista através do conjunto de funções intituladas como *"send notification functions"*. Estas funções comunicam-se através de eventos distribuídos em tópicos do SNS e filas SQS para realizar o envio de notificações, o qual é feito através da comunicação com APIs de provedores. Além disso, para emitir eventos agendados, elas utilizam o EventBridge Scheduler. Este fluxo está descrito em detalhes na Seção 5.5.3.2.

Por fim, o serviço de *Notification* pode ser entendido como uma API REST com rotas voltadas para o gerenciamento do ambiente do sistema de notificações e outras rotas voltadas para que *softwares* de usuários se integrem ao sistema de notificações. Inter-

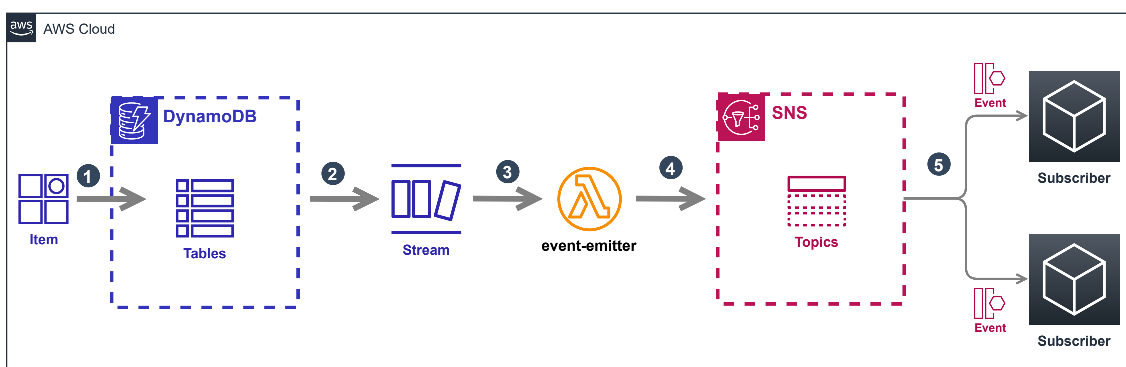
namente o serviço utiliza intensamente a comunicação assíncrona através publicação e consumo de eventos para as notificações serem enviadas e entregues aos destinatários.

5.5.3.1 Transactional Outbox

É comum que sistemas necessitem realizar operações no banco de dados e emitir eventos ao final de um processamento. Entretanto, uma transação de bancos de dados e uma operação de emitir eventos através de um sistema de mensageria são operações distintas que por padrão não ocorrem de forma atômica. Por não ocorrerem de forma atômica, o sistema é suscetível a inconsistências, visto que uma modificação poderia ocorrer no banco de dados, mas o evento associado a esta modificação poderia não ser emitido. O contrário também seria possível.

Inconsistência ocasionadas devido à impossibilidade de realizar ambas operações de forma atômica podem trazer problemas para aplicações que necessitam de garantias como: se uma modificação foi realizada no banco de dados, o evento associado foi emitido para os interessados. A ocorrência de inconsistências como essas pode ocasionar o não envio de notificações para o caso do sistema desenvolvido neste trabalho. Por exemplo, se o evento de que uma nova carga de trabalho está disponível não for emitido no momento em que um *Workload* for salvo no banco de dados, a função responsável por processar a carga de trabalho assincronamente nunca terá conhecimento do novo *Workload*, visto que nunca será invocada pelo sistema de mensageira. Isso faria com que todas as notificações envolvidas no *Workload* não fossem enviadas, o que seria péssimo para todos os usuários e destinatários do sistema de notificações.

Figura 5.15: Representação do padrão *Transactional Outbox* utilizando serviços da AWS conforme implementado no sistema de notificações.



Para resolver esse problema, podemos utilizar o padrão conhecido como *Transactional Outbox*. De acordo com Bittencourt (2020), ao usar o padrão o *Transactional*

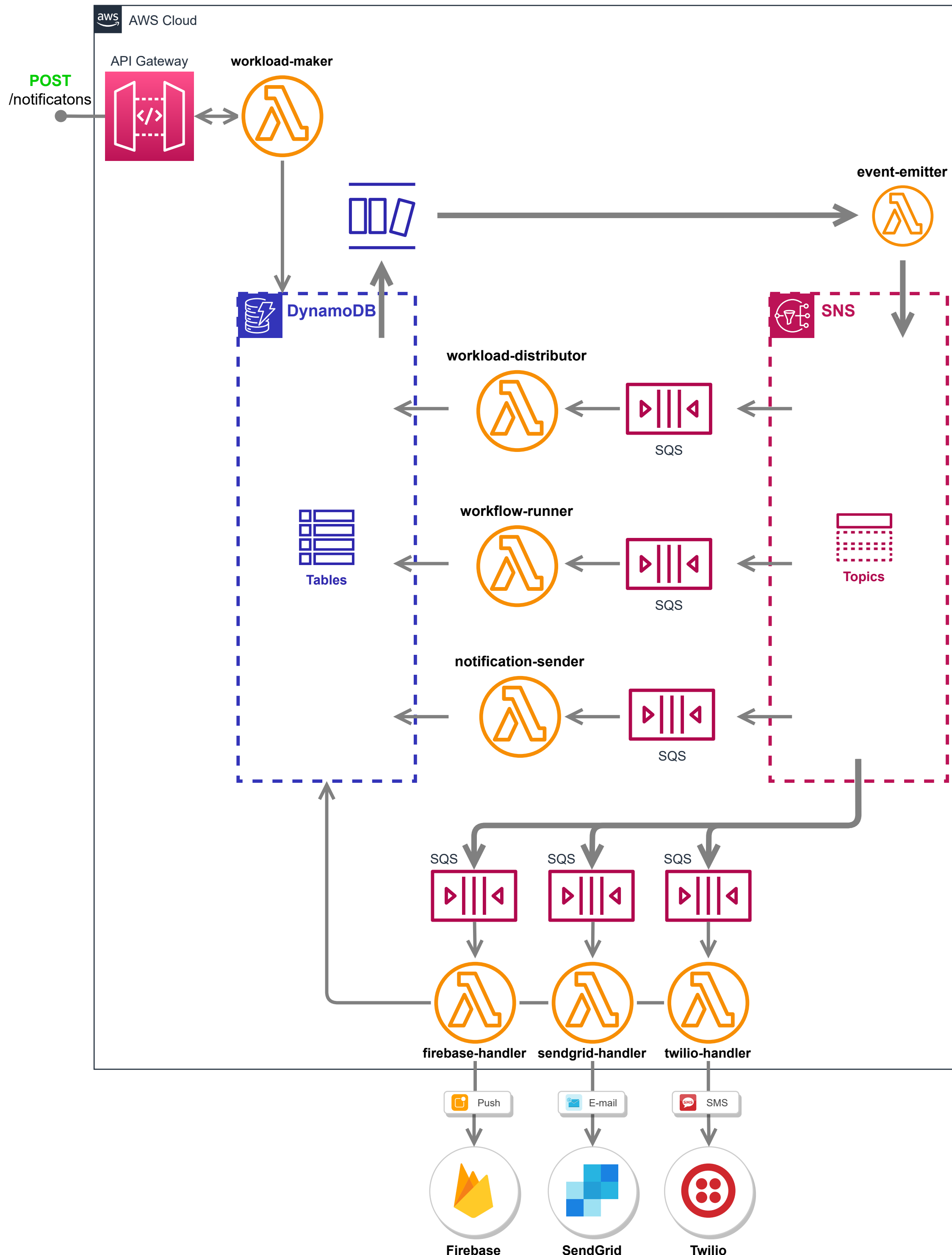
Outbox, durante o processamento utiliza-se apenas uma das ferramentas: o banco de dados. Assim, garantimos que a operação será atômica e que não existirão inconsistências. Posteriormente, de maneira assíncrona, por meio de um mecanismo de *push* ou *pull*, as modificações que ocorreram no banco de dados são consumidas e os eventos associados são emitidos.

Assim, na Figura 5.15 podemos visualizar a implementação do padrão *Transactional Outbox* utilizando o DynamoDB Streams, uma função do AWS Lambda chamada `event-emitter` e tópicos do SNS. Toda mudança ocorrida no DynamoDB é disponibilizada no DynamoBD Streams, o qual invoca a função `event-emitter` que é responsável por compreender as alterações e emitir os eventos adequados através de tópicos do SNS. Esse padrão foi amplamente utilizado para adicionar garantias de consistências durante o envio de notificações.

5.5.3.2 Fluxo de envio de notificação

Na parte inferior da Figura 5.14 visualizamos de maneira simplificada o funcionamento do envio de notificações através do sistema de notificações. Nesta Seção este fluxo é descrito em detalhes com o auxílio da Figura 5.16.

Figura 5.16: Detalhes da arquitetura do serviço *Notification* para tarefa de envio de notificações.

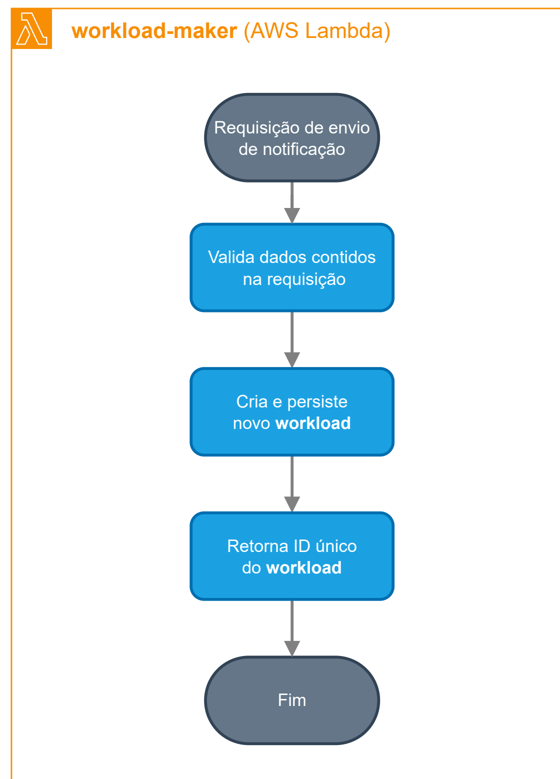


Após o ambiente do sistema de notificações de um usuário estar configurado com *templates*, *workflows*, *recipients* e integrações com provedores, o envio de notificações pode ocorrer através da comunicação entre o *software* dos usuários com o sistema de

notificações. Para realizar o envio, uma requisição HTTP deve ser enviada para a rota `POST /notifications` da API do serviço de *Notification*, a qual é exposta através do API Gateway.

O API Gateway, ao receber uma requisição autorizada nesta rota, invoca a função `workload-maker` do serviço de *Notification*. Essa função é responsável por tratar a requisição de envio de maneira que uma carga de trabalho (*Workload*) seja criada. Após a criação da carga de trabalho, o identificador único do *Workload* é retornado para *software* integrado ao sistema de notificações. O processamento da carga de trabalho ocorre de maneira totalmente assíncrona. Na Figura 5.17 é possível visualizar o fluxograma das ações executadas por esta função do AWS Lambda.

Figura 5.17: Fluxograma das operações realizadas pela função *workload-maker* do serviço de *Notification*.

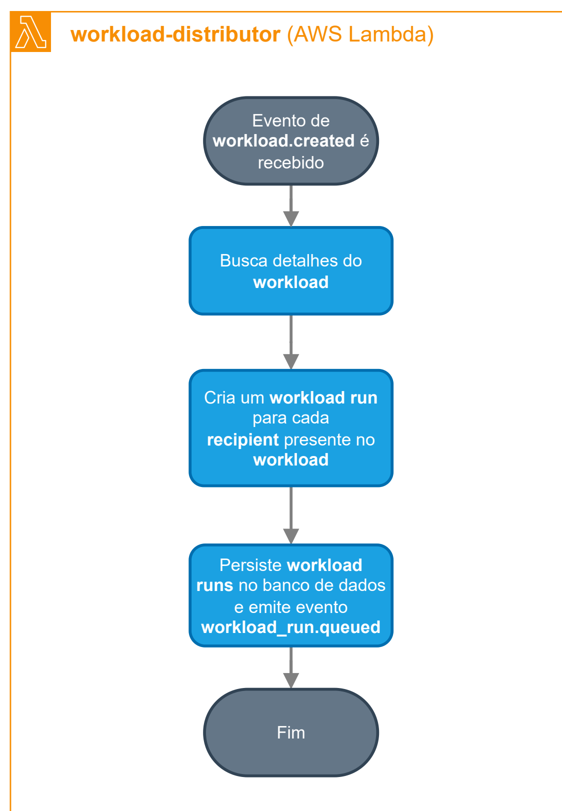


Após persistir o *Workload* no DynamoDB, o padrão *Transactional Outbox* é utilizado. A inserção no banco de dados é capturada pelo DynamoDB Streams, o qual invoca a função `event-emitter` que formata o evento e o emite através de um tópico do SNS. O evento `workload.created` é recebido pela função `workload-distributor` através de uma fila do SQS, pois esta função está inscrita no tópico associado ao evento.

A função `workload-distributor` é responsável por distribuir a carga de trabalho associada a um *Workload* em cargas menores e específicas para cada destinatário.

Assim, a carga de trabalho para cada destinatário pode ser processada paralelamente por instâncias de funções do AWS Lambda. Para distribuir a carga, uma entidade de *Workload Run* é criada e persistida no DynamoDB para cada um dos destinatários contidos na carga de trabalho total. As entidades de *Workload Run* criadas no processo descrito na Figura 5.18 emitem o evento de `workload_run.queued` através do padrão de *Transactional Outbox* e são processadas assincronamente por outras funções do AWS Lambda.

Figura 5.18: Fluxograma das operações realizadas pela função *workload-distributor* do serviço de *Notification*.

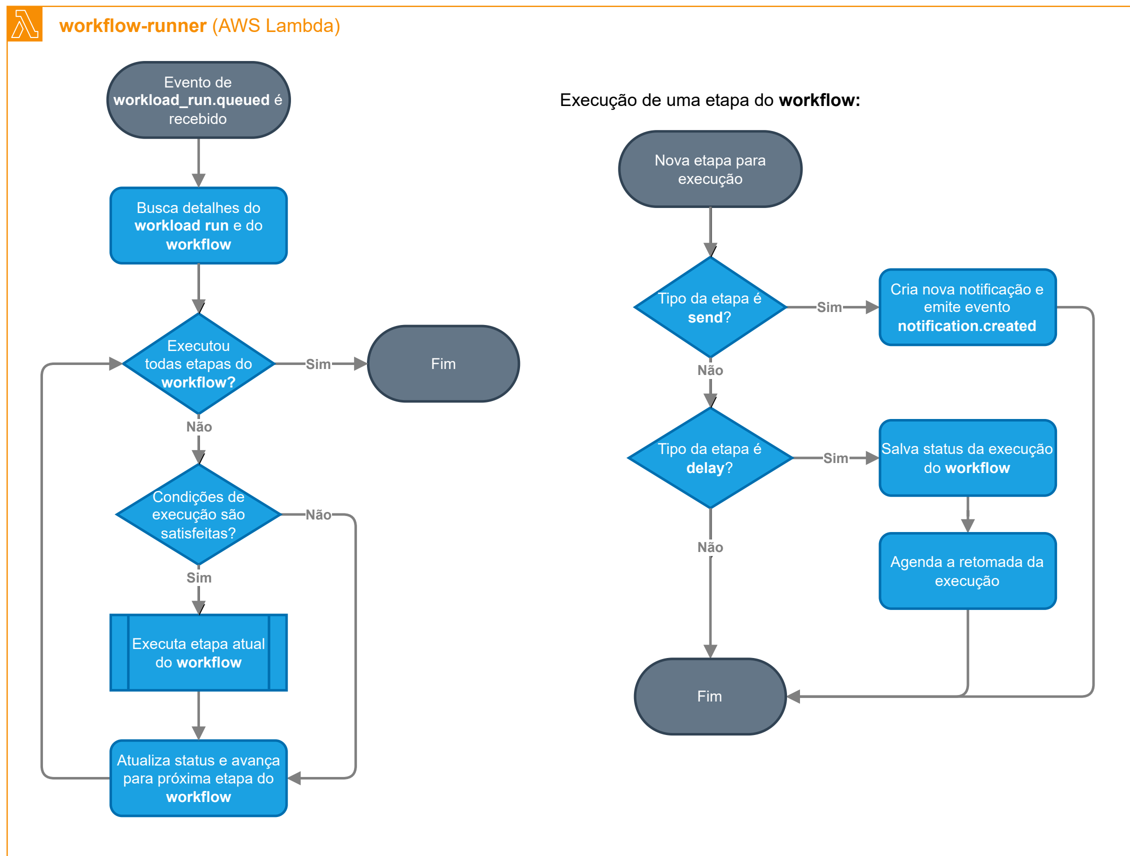


As funções `workflow-runner` estão inscritas no tópico adequado para serem invocadas ao receberem o evento `workload_run.queued`. Assim, após a distribuição das cargas de trabalho, estes eventos são recebidos através de uma fila SQS para que a carga de trabalho seja executada por meio de um *Workflow*.

O *Workflow* executado para um determinado *Workload Run* é definido de acordo com o tipo de *Workload* que criado pelo usuário através da API. Caso o identificador de um *Workflow* tenha sido indicado na requisição, este *Workflow* é carregado do banco de dados. Já nos casos em que a forma de envio na requisição é `simple`, `element` ou `template`, o *Workflow* executado é definido em tempo de execução e contém apenas uma etapa: o envio de uma notificação.

O processo de execução de um *Workflow* para uma carga de trabalho é bastante

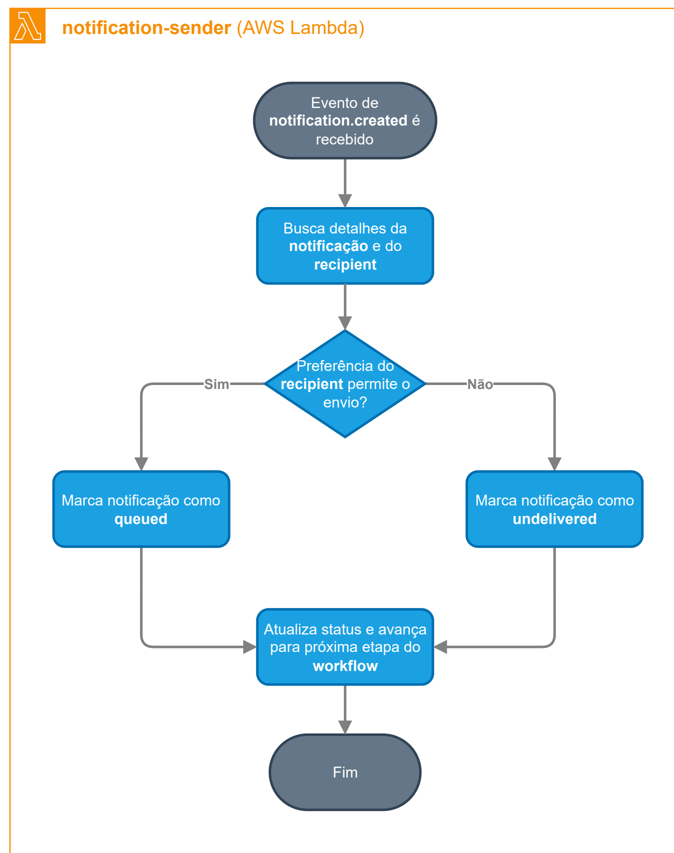
Figura 5.19: Fluxograma das operações realizadas pela função *workflow-runner* do serviço de *Notification*.



simples e está descrito no fluxograma da Figura 5.19. As etapas contidas no *Workflow* são percorridas sequencialmente até que todas as etapas sejam executadas ou até que ocorra alguma pausa na execução. Antes de executar cada uma das etapas, verificam-se as condições associadas à etapa para decidir se ela deve ser executada ou não. As etapas que não satisfazem todas as condições impostas são ignoradas. Caso a etapa em execução seja uma etapa de envio, uma notificação é persistida no banco de dados e, consequentemente, um evento de `notification.created` é emitido. Já para os casos em que a etapa em execução é uma etapa de *delay*, o processamento do *Workload Run* é interrompido e agendado para ser retomado no futuro (o processo de agendamento está descrito em detalhes na Seção 5.5.3.4. Por fim, o estado do *Workload Run* é persistido no DynamoDB ao final de cada etapa.

Conforme comentado, durante a execução de etapas de envio, eventos do tipo `notification.created` são emitidos. Estes eventos são recebidos pelas funções de `notification-sender` através de filas SQS e tópicos do SNS. Estas funções são responsáveis por verificar se o processamento de uma notificação deve continuar ou não.

Figura 5.20: Fluxograma das operações realizadas pela função *notification-sender* do serviço de *Notification*.

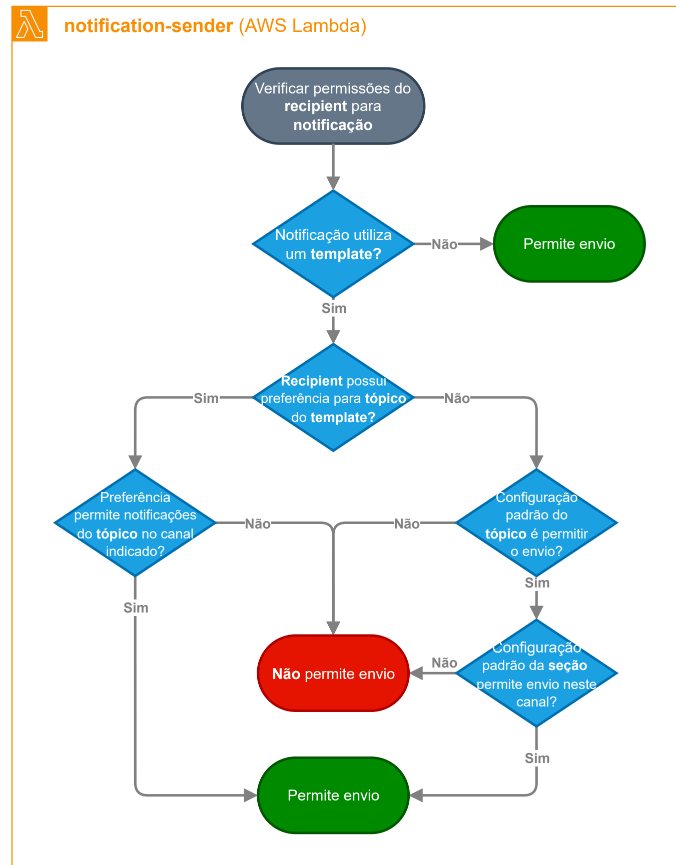


Uma notificação deve ser enviada caso as preferências do destinatário indicarem que ele possui interesse no conteúdo dela. Para isso, primeiramente verificamos se a notificação possui um *template* vinculado, pois preferências só são avaliadas para notificações com conteúdo previamente definido, ou seja, caso a notificação não inclua um *template*, ela sempre é enviada ao destinatário. Nos casos em que há um *template*, avaliamos as preferências personalizadas do destinatário para o tópico vinculado ao *template*. Caso não existam personalizações, as configurações padrão do tópico e da seção de preferências são utilizadas. O processo de avaliação das preferências pode ser visualizado em detalhes na Figura 5.21.

Para os casos em que o destinatário optou por receber o conteúdo da notificação, o status dela irá transicionar de `created` para `queued` e, consequentemente, o evento `notification.queued` é emitido. Porém, nos casos que o destinatário não possui interesse no conteúdo da notificação, o status é modificado para `undelivered` e emite-se um evento de `notification.undelivered`. Neste último caso, não existe a possibilidade do status ser alterado futuramente para `queued`.

Nos dados dos eventos de `notification.queued` existem campos que identifi-

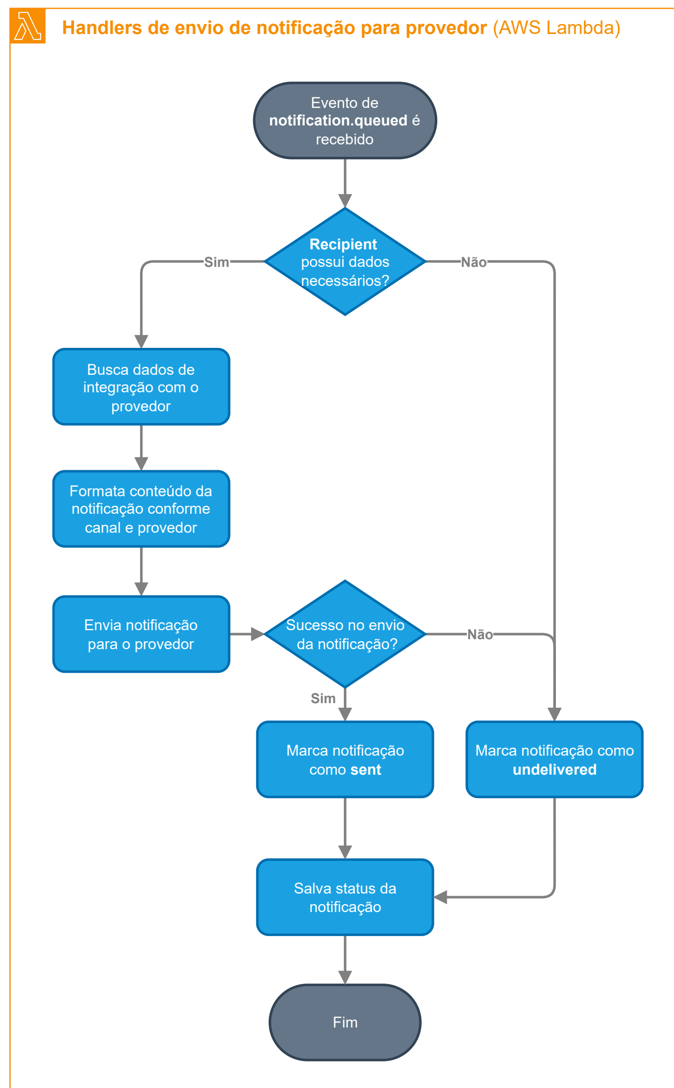
Figura 5.21: Fluxograma das operações realizadas para validar se as preferências do destinatário permitem o envio da notificação.



cam o canal e o provedor que são utilizados para enviar a notificação. Estas informações são utilizadas pelo SNS para filtrar o evento e destiná-lo para fila SQS da função adequada do AWS Lambda. Por exemplo, no caso de uma notificação de SMS através do provedor Twilio, o evento é recebido pela função `twilio-handler`. As funções que realizam o envio da notificação para o provedor são chamadas de *handlers*. A responsabilidade destas funções é converter o conteúdo da notificação, normalmente baseado em *elements*, para o formato adequado e esperado pelo provedor. Estas funções conhecem os detalhes da comunicação de cada provedor e os abstraem para o restante do sistema. Importante ressaltar que nesta etapa do processamento de uma notificação, é necessário os dados de integração do usuário com o provedor, por isso as integrações com provedores devem estar devidamente configuradas.

Após as funções *handlers* realizarem a comunicação com os provedores, o status da notificação é atualizado para representar o sucesso ou não do envio. Caso a comunicação tenha sido um sucesso e a notificação foi enviada para o provedor, o status da notificação é alterado para `sent` e o evento `notification.sent` é emitido. Porém, caso ocorram falhas, o status salvo no DynamoDB é o `undelivered` e o evento

Figura 5.22: Fluxograma das operações realizadas por funções de integração com provedores de envio de notificação.



`notification.undelivered` é emitido. Nesta etapa do processamento, os detalhes da comunicação com o provedor são persistidos para auxiliar em tarefas de auditoria e *debugging*.

Finalmente, a atividade de processamento de uma carga de trabalho e envio de notificação está finalizada. De maneira resumida, podemos descrevê-la como uma requisição de carga de trabalho, a qual é processada assincronamente. A carga de trabalho total é dividida em partes menores especificas para cada destinatário. Estas partes são processadas através de *Workflows* previamente construídos ou definidos em tempo de execução. Cada uma das etapas de um *Workflow* pode realizar o envio de uma notificação para o destinatário ou interromper o processamento para continuá-lo no futuro. O envio das notificações é dependente das preferências do usuário e processado por funções *handler*, as quais se integram aos provedores.

5.5.3.3 Eventos dos provedores

Na Seção 5.5.3.2, vimos como é realizado o envio de notificações para os provedores. Em caso de sucesso no envio, o evento `notification.sent` é emitido. Entretanto, como vimos na Figura 5.12, uma notificação pode conter *status* posteriores: `delivered` para indicar que a notificação foi entregue ao destinatário e `seen` para indicar que o destinatário visualizou a notificação. Os eventos que geram estes *status* tem origem no provedor e, por isso, o sistema de notificações precisa encontrar maneiras de acessar tais informações. Alguns provedores não possuem suporte a tais eventos, às vezes devido a limitações do próprio canal e outras vezes devido à ausência de suporte do próprio provedor.

Assim, para permitir que os usuários do sistema de notificações tenham conhecimento destes eventos após enviarem as notificações, foram implementadas integrações via *webhooks* com os provedores para que eles informem ao sistema as modificações ocorridas no *status* das notificações. Consequentemente, os usuários agora podem ter acesso a esses dados valiosos para análise do comportamento dos destinatários.

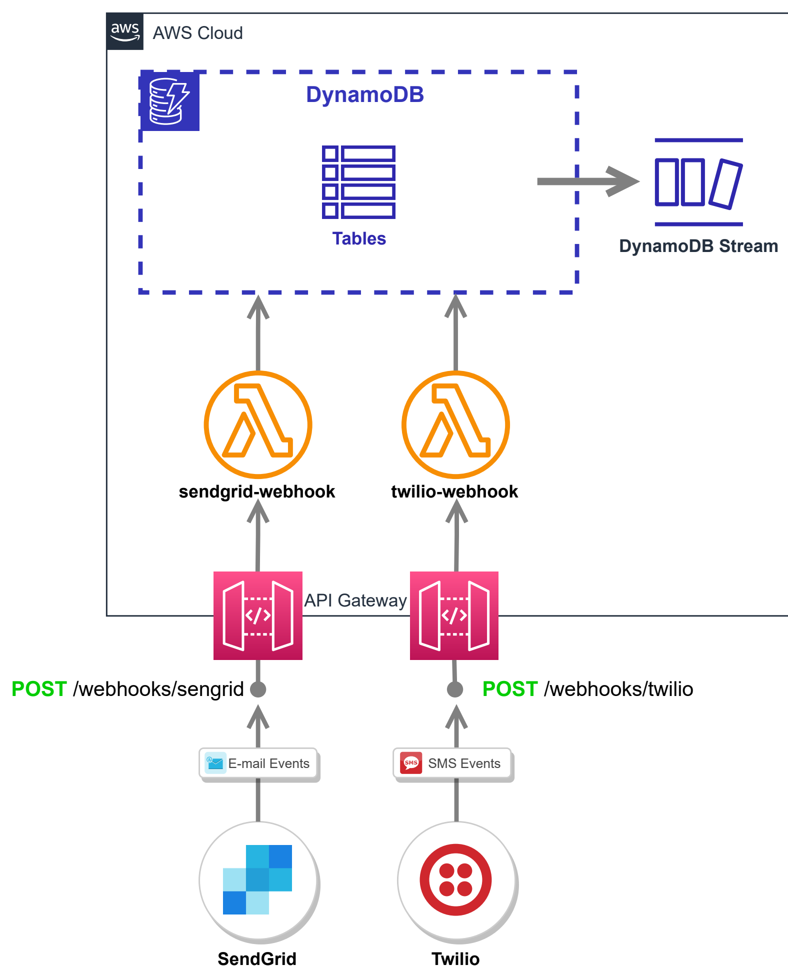
Para isso, a comunicação com os provedores ocorre através de requisições HTTP para rotas específicas para cada provedor. Na Figura 5.23 é possível visualizar como a comunicação entre provedor e sistema de notificações ocorre para informar a ocorrência de eventos. Antes da comunicação ocorrer, é necessário que o sistema de notificações seja configurado como destino dos eventos nas opções do provedor. Após esta configuração, todo evento vinculado às notificações enviadas através do sistema de notificações são repassadas a ele através de requisições HTTP.

O API Gateway, ao receber tais requisições, as encaminha para a função adequada para o provedor. Estas funções reconhecem o tipo de evento recebido e, então, o armazena no banco de dados DynamoDB. Os eventos registrados no banco de dados são distribuídos através do padrão *Transactional Outbox* com a utilização do DynamoDB Streams e SNS e, ficam disponíveis para que os serviços do sistema de notificações tratem os eventos da forma adequada.

5.5.3.4 Interrupção e agendamento de Workload Runs

Durante a execução do *Workflow* associado a um *Workload Run*, etapas de *delay* podem causar a interrupção da execução para que ela seja continuada em um momento futuro. Isso é possível através da adoção de estratégias de agendamento de eventos, as

Figura 5.23: Detalhes de como é feito o recebimento de eventos dos provedores.



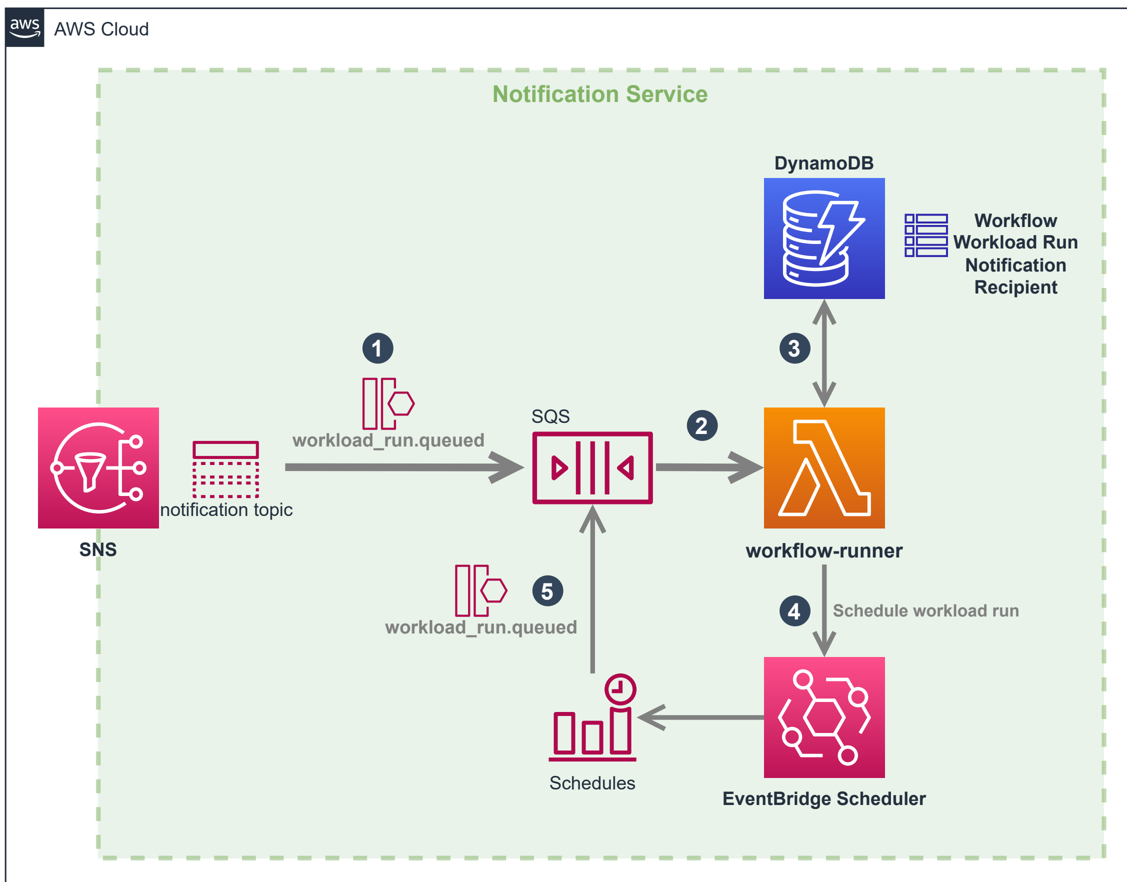
quais são descritas nesta Seção juntamente com a arquitetura da solução adotada para permitir os agendamentos.

Primeiramente, durante a construção de *Workflows*, pode-se utilizar etapas de *delay* para definir a estratégia de agendamento que é usada caso as condições da etapa sejam satisfeitas. Há duas estratégias de agendamento:

- `until`: defini-se a data exata em que a execução do *Workflow* deve ser retomada no formato `yyyy-MM-dd HH:mm:ss`;
- `duration`: nesta estratégia de agendamento, define-se a duração do *delay*, ou seja, a duração do período de interrupção da execução do *Workflow*. A duração é definida por uma unidade de tempo e o valor da duração na unidade de tempo indicada. As unidades de tempo disponíveis são `seconds`, `minutes`, `hours` e `days`.

Após a construção de *Workflows* com etapas de *delay*, a interrupção e o agendamento da execução é feita pela função `workflow-runner` conforme descrito na Figura 5.24.

Figura 5.24: Detalhes de como é feita a execução de uma etapa de *delay* de um *Workflow*.



Inicialmente, um *Workload Run* é criado com a função `workload-distributor`, conforme demonstrado na Figura 5.16. Consequentemente, um evento de `workload_run.queued` é emitido e recebido na função `workflow-runner` através de uma fila SQS. Durante o processamento do *Workflow*, a função utiliza o DynamoDB para buscar os dados necessários, atualizar o estado de execução e executar de fato a etapa.

No momento da execução de uma etapa de *delay*, a função verifica qual a forma mais adequada para realizar o agendamento visando obter a melhor performance e precisão. O serviço Amazon EventBridge Scheduler, usado para realizar agendamentos, possui garantias de precisão na unidade de minutos. Entretanto, mensagens adicionadas em filas SQS possuem um parâmetro de *delay* dado em segundos e com valor máximo de 900 segundos (15 minutos). Então, optou-se por escolher estratégias de agendamento diferentes dada a duração da interrupção para permitir uma maior precisão em agendamentos de curta duração (inferior a 15 minutos).

Assim, durante a execução de uma etapa de *delay*, verificamos se a duração da interrupção é inferior a 15 minutos. Nestes casos, o agendamento é feito através da fila

SQS presente na Figura 5.24 ao utilizar o valor de *delay* adequado para o agendamento. Já para os casos em que a duração é superior a 15 minutos, entende-se que imprecisões de alguns segundos são toleráveis e, por isso, utiliza-se o serviço Amazon EventBridge Scheduler para criar um evento agendado na data e hora adequada para a retomada da execução do *Workflow*. Assim, em ambos os casos, após o fim da duração da interrupção, uma mensagem é disponibilizada na fila SQS e a função `workflow-runner` continua a execução do *Workflow* na etapa seguinte.

5.5.4 Rotas da API

Nesta Seção, são descritas as rotas existentes na API do serviço de *Notification*, bem como o formato dos dados, o método de autenticação e autorização necessário e as respostas possíveis para cada uma das rotas. Dado o alto número de rotas de gerenciamento contidas neste serviço, alguns detalhes de rotas de menor relevância são omitidos. Entretanto, todos os detalhes podem ser consultados no Anexo B.

5.5.4.1 Autenticação e Autorização

As rotas presentes na API do serviço de *Notification* possuem métodos de autenticação e autorização diferentes conforme a finalidade da rota. Para rotas de gerenciamento do ambiente do sistema de notificações, utilizou-se o método descrito na Figura 5.8 baseada na integração com o Amazon Cognito, visto que essas rotas foram construídas com o objetivo de serem utilizadas por uma aplicação *front-end*. Já para rotas destinadas ao envio de notificações e que são as rotas utilizadas por outras aplicações de *software* para se integrar ao sistema de notificações, o método de autenticação e autorização escolhido é baseado na utilização das chaves de acesso (*API Keys*) gerenciadas pelo serviço de IAM conforme descrito na Figura 5.9.

Assim, em cada um das rotas descritas nesta Seção, é indicado qual o modelo de autenticação e autorização deve ser utilizado.

5.5.4.2 Gerenciamento de integrações

Para que notificações sejam enviadas através de um canal de distribuição, é necessário que a *Organization* possua integrações com provedores configuradas. Através destas configurações, o sistema de notificações terá as informações necessárias para conseguir

se comunicar com a API do provedor. Logo, para cada provedor, o conjunto de informações necessárias na configuração é diferente. Assim, para gerenciar estas integrações, existem rotas para consultar, criar e deletar integrações, as quais utilizam como método de autenticação e autorização a integração com o Amazon Cognito conforme descrito na Figura 5.8. A seguir são apresentadas as descrições destas rotas.

Consultar detalhes de uma integração: (`GET /integrations/:id`) esta rota é utilizada para consultar, através do identificador único, os detalhes de uma integração previamente configurada. Em caso de sucesso, a rota responde com *status code 200 OK* e um objeto JSON contendo os detalhes da configuração através dos dados de uma entidade *Integration* descrita na Seção 5.5.2. Por fim, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Configurar nova integração: (`POST /integrations`) rota utilizada para configurar uma nova integração. Para isso deve ser fornecido o provedor desejado, o canal desejado e os dados específicos para aquele provedor. Em caso de sucesso, a rota responde apenas com *status code 201 Created.* Finalmente, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

Deletar integração: (`DELETE /integrations/:id`) rota utilizada para deletar uma integração existente através de seu identificador único. Em caso de sucesso, a rota responde apenas com *status code 200 Ok.* Entretanto, a API responde com o *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3 se o identificador não existir.

Os detalhes do formato dos dados necessários para a integração com cada um dos provedores suportados, bem como exemplos de requisições e respostas, podem ser consultados na documentação presente no Anexo 5.3.

5.5.4.3 Gerenciamento de templates

Para o gerenciamento de *templates* de notificações, os quais podem ser reutilizados para enviar múltiplas notificações, o serviço de *Notification* oferece rotas que permitem a consulta, criação, atualização e remoção de *templates*. Estas rotas utilizam o método de autenticação e autorização descrito na Figura 5.8 e estão descritas brevemente a seguir.

Consultar detalhes de um *template*: (`GET /templates/:id`) permite que detalhes de um *template* sejam consultados através do identificador único. Em caso de

sucesso, a rota responde com *status code 200 OK* e um objeto JSON contendo os detalhes do *template* e seus blocos de *elements* conforme descritos na Seção 5.5.2. Por fim, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Criar novo *template*: (`POST /templates`) rota utilizada para criar um novo *template* através da definição de seu nome e *layout* para cada canal de distribuição desejado através de blocos de conteúdo (*elements*). Em caso de sucesso, a rota responde com o*status code 201 Created* e o identificador do *template*. Todavia, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

Atualizar *template*: (`PUT /templates/:id`) rota utilizada para atualizar as definições de *layouts* de um *template* existente. A requisição deve conter os dados no mesmo formato que a rota de criação de um novo *template*. Se houver sucesso, a rota responde apenas com *status code 200 OK*. Contudo, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request*. Da mesma forma, se o identificador único não existir, a API responde com *status code 404 Not Found*. Assim, como nas demais rotas, as mensagens de erro estão no formato descrito na Seção 5.3.

Deletar *template*: (`DELETE /templates/:id`) rota utilizada para deletar um *template* existente através de seu identificador único. Em caso de sucesso, a rota responde apenas com *status code 200 Ok*. Entretanto, a API responde com o *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3 se o identificador não existir.

A documentação presente no Anexo 5.3 pode ser consultada para visualizar os detalhes dos dados de entrada e saída de cada rota, bem como exemplos e descrições completas.

5.5.4.4 Gerenciamento de workflows

As tarefas de gerenciamento dos *workflows* vinculados a uma *Organization* podem ser realizadas através das rotas de consulta, criação, atualização e exclusão descritas a seguir. Estas rotas são autorizadas e autenticadas através da integração com o Amazon Cognito no processo descrito na Figura 5.8.

Consultar detalhes de um *workflow*: (`GET /workflows/:id`) esta rota permite que detalhes de um *workflow* sejam consultados através de seu identificador único. Se não houverem erros, a rota responde com *status code 200 OK* e um objeto JSON contendo

os detalhes do *workflow* e de suas etapas de execução conforme foram descritas na Seção 5.5.2. Entretanto, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Criar novo *workflow*: (`POST /workflows`) esta rota realiza a criação de um novo *workflow* através da definição de seu nome. As etapas do *workflow* são definidas através de um processo de atualização das definições do *workflow*. Se houver sucesso, a rota responde apenas com o *status code 201 Created* e o identificador único do *workflow*. Todavia, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

Atualizar *workflow*: (`PUT /workflows/:id`) através desta rota é possível atualizar os dados de um *workflow* e definir suas etapas de execução após a criação do mesmo. A requisição deve conter as etapas desejadas para o *workflow*. Em caso de sucesso, a rota responde apenas com *status code 200 OK*. Entretanto, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request*. Já para os erros relacionados ao identificador único não existir, a API responde com *status code 404 Not Found*. Da mesma forma que nas demais rotas, as mensagens de erro estão no formato descrito na Seção 5.3.

Deletar *workflow*: (`DELETE /workflows/:id`) permite que um *workflow* existente seja deletado através de seu identificador único. Em caso de sucesso, a rota responde apenas com *status code 200 Ok*. Todavia, se o identificador único não existir, a API responde com o *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

No Anexo B, encontram-se os detalhes do formato aceito e retornado por cada uma destas rotas, bem como exemplos e descrições detalhadas.

5.5.4.5 Gerenciamento de preferências

Para gerenciar as preferências que os destinatários possuem acesso para optar ou não por receber, o serviço de *Notification* implementa um conjunto de rotas para consultar, criar e deletar seções e tópicos de preferências. Estas rotas também são voltadas para serem utilizadas por uma aplicação *front-end*, por isso utiliza-se a integração com o Amazon Cognito para realizar a autorização das requisições conforme descrito na Figura 5.8. A seguir descreveremos brevemente as rotas existentes para o gerenciamento de preferências.

Consultar uma seção: (`GET /preferences/sections/:id`) através do iden-

tificador único da seção, permite que seus detalhes sejam consultados através de seu identificador único. Em caso de sucesso, a rota responde com *status code 200 OK* e um objeto com os detalhes da entidade *Preference Section* no formato JSON. Todavia, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Criar nova seção: (`POST /preferences/sections`) através da definição do nome da seção e de suas opções de canais, permite que uma nova seção seja cadastrada. Se houver sucesso, a rota responde com o *status code 201 Created* e o identificador único da seção. Entretanto, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3 se houver algum erro de validação na requisição.

Deletar seção: (`DELETE /preferences/sections/:id`) através de um identificador único, permite que uma seção seja deletada. Em caso de sucesso, a rota responde apenas com *status code 200 Ok*. Já para os casos em que o identificador único não existir, a API responde com o *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Consultar um tópico: (`GET /preferences/topics/:id`) permite que os detalhes de um tópico de preferências seja consultado. O formato dos dados de entrada, saída e erros são os mesmo descritos para a rota de consulta de seções.

Criar novo tópico: (`POST /preferences/topics`) dado o nome do tópico, a configuração padrão por optar ou não o envio, o identificador seção e uma lista de *templates*, um novo tópico de preferências pode ser cadastrado para uma seção especifica. Os casos de sucesso e falha são os mesmos descritos para a rota de criação de seção, porém a adição da validação de que a seção e os *templates* citados existem.

Deletar tópico: (`DELETE /preferences/topics/:id`) a partir do identificador único do tópico, realiza a exclusão do tópico. O formato das respostas e erros são os mesmos descritos para a rota de exclusão de seção.

Para consultar todos os detalhes do funcionamento, formatos dos dados de entrada e saída e exemplos, as documentações da API presentes no Anexo B podem ser acessadas.

5.5.4.6 Gerenciamento de destinatários

Antes de realizar o envio de notificações para um destinatário, é necessário que eles estejam cadastrados no sistema de notificações. Para auxiliar no controle dos destinatários vinculados a uma *Organization*, existem rotas de gerenciamento de destinatários. Isto é, a consulta de destinatários cadastrados, o registro de novos destinatários, a atualiza-

ção de suas informações e sua exclusão. Além disso, é possível gerenciar as preferências dos destinatários ao optarem ou não receberem notificações atreladas a tópicos de preferências. Visto que este controle é realizado por outra aplicação de *software* integrada à API de envio do sistema de notificações, o modo de autenticação utilizados nestas rotas é baseado na utilização de chaves de acesso conforme descrito na Figura 5.9.

Consultar detalhes de um destinatário: (`GET /recipients/:id`) esta rota permite que detalhes de um destinatário sejam consultados através de seu identificador único. Se não houverem erros, a rota responde com *status code 200 OK* e um objeto JSON contendo os detalhes todos os dados do destinatário conforme foram descritas na entidade de *Recipient* da Seção 5.5.2. Entretanto, se o identificador fornecido não existir, a API responde com *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Criar e atualizar destinatário: (`PUT /recipients/:id`) esta rota é usada para cadastrar ou atualizar destinatários a fim de facilitar o gerenciamento. Caso o destinatário ainda não exista, ele é registrado. Caso contrário, seus dados são atualizados. Assim, esta rota recebe obrigatoriamente o identificador único escolhido para o destinatário, bem como seus dados de integração e de perfil. Se houver sucesso, a rota responde apenas com o *status code 200 OK*. Todavia, se houver algum erro de validação na requisição, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

Deletar destinatário: (`DELETE /recipients/:id`) permite que um destinatário existente seja deletado através de seu identificador único. Em caso de sucesso, a rota responde apenas com *status code 200 OK*. Todavia, se o identificador único não existir, a API responde com o *status code 404 Not Found* e uma mensagem de erro no formato descrito na Seção 5.3.

Preferências do destinatário:

(`PUT /recipients/:id/preferences/:topicId`) através desta rota é possível gerenciar as preferências de um destinatário para um tópico de preferências específico através de seus identificadores únicos. Nos dados da requisição devem ser informados os canais preferidos do destinatário e a sua opção por receber ou não notificações. Em caso de sucesso, a rota responde apenas com o *status code 200 OK*. Entretanto, em caso de erro de validação na requisição, a API responde com *status code 400 Bad Request* e uma mensagem de erro no formato descrito na Seção 5.3.

As rotas de gerenciamento de preferências podem ser encontradas em detalhes nas

documentações presentes no Anexo B.

5.5.4.7 Envio de notificações

Para enviar notificações, há apenas a rota `POST /notifications`, a qual pode receber solicitações de envio dos quatro tipos descritos na Seção 5.5.2. A escolha do tipo de envio é feita a partir do formato dos dados presentes na requisição e são descritos a seguir. Ademais, visto que esta rota é utilizada por outras aplicações de *software*, o método de autenticação e autorização aplicado é baseado na utilização das chaves de acesso da *Organization* conforme descrevemos na Figura 5.9.

No objeto JSON presente no corpo da requisição para esta rota, devem ser fornecidos dois campos: `to` e `message`. O primeiro trata-se de uma lista de identificador únicos de destinatários para os quais deseja-se enviar notificações. Já o segundo contém um objeto com campos descrevendo o tipo de envio desejado e o conteúdo da notificação de acordo com o tipo selecionado e conforme as definições de cada tipo de envio feitas na entidade de *Workload* da Seção 5.5.2.

Dado que, conforme descrito na Seção 5.5.3.2, o envio ocorre de forma totalmente assíncrona, em casos de sucesso, a API retorna uma resposta contendo o *status code 201 Accepted* e o identificador único do *workload*. Este identificador, pode ser utilizado posteriormente para consultar as notificações envidas através desta solicitação. Entretanto, se houverem erros de validação, a API retorna uma resposta com o *status code 400 Bad Request* e mensagens de erro no formato descrito na Seção 5.3.

Os detalhes de como é realizado o envio de notificações através desta rota podem ser consultados nas documentações da API presentes no Anexo B.

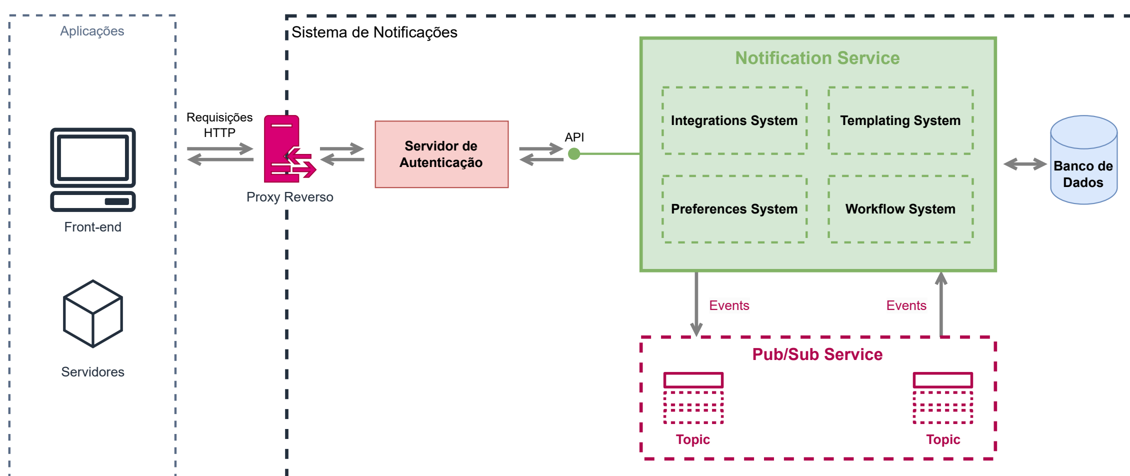
5.6 Módulo de *Analytics Service*

Esta Seção tem como objetivo descrever em detalhes o funcionamento do módulo de *Analytics*. Para isso, é apresentada a visão geral do funcionamento do serviço, bem como os detalhes de sua arquitetura e API REST.

5.6.1 Visão Geral

O serviço de *Analytics* é responsável por armazenar, analisar e disponibilizar os dados relacionados a todo envio realizado pelo sistema de notificações. As informações geradas durante o envio de notificações e durante o engajamento dos destinatários com as notificações possuem grande valor agregado e podem auxiliar no processo de tomada de decisão dos usuários do sistema. Logo, podemos definir uma arquitetura de alto nível para o serviço conforme descrito na Figura 5.25, na qual podemos visualizar a necessidade de um sistema de *proxy* reverso, servidor de autenticação, sistema de *pub/sub* distribuído para recebimento de eventos e banco de dados para armazenamento e consulta de informações.

Figura 5.25: Arquitetura de alto nível do serviço de *Analytics* do sistema de notificações.

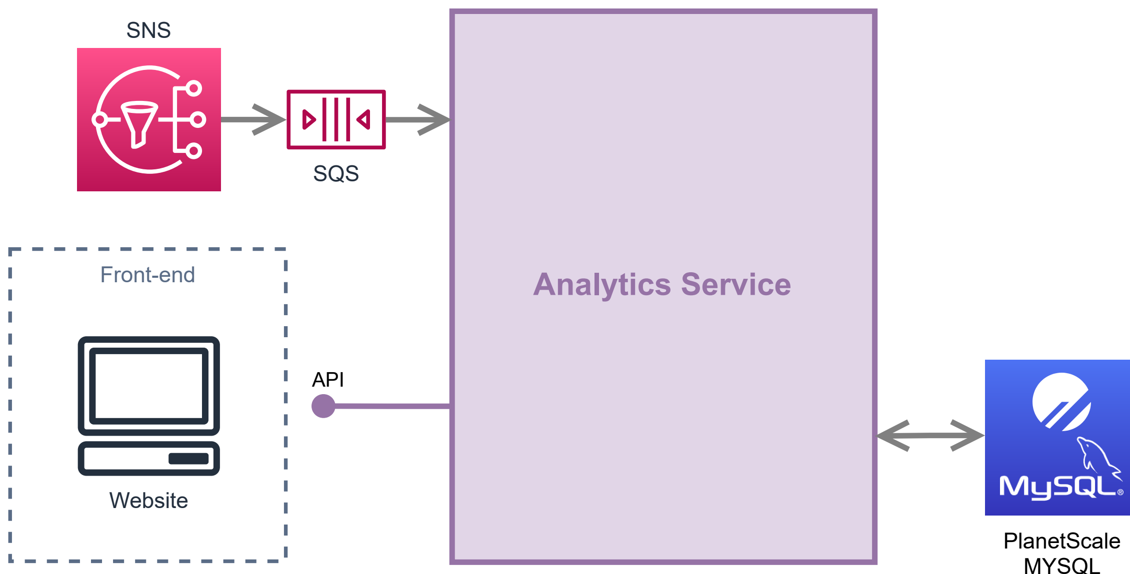


Durante o envio de notificações através do sistema construído neste trabalho, toda etapa de processamento de notificações e *Workflows* é armazenada através de eventos e processos de replicação de dados. Assim, os usuários do sistema possuem os dados históricos de toda notificação enviada e, possivelmente, de toda interação dos destinatários com as notificações. Estes dados são importantes durante o desenvolvimento da integração, monitoramento em ambiente de produção e refinamentos de acordo com o comportamento dos destinatários.

Logo, os usuários do sistema de notificações podem visualizar o histórico de notificações enviadas e de *Workflows* executados, bem como os eventos emitidos em cada etapa de processamento e possíveis erros ocorridos. Inicialmente, isso serve de insumo para realizar tarefas de *debugging* da integração com o sistema de notificações. Entretanto, com a utilização do sistema em ambiente de produção, os usuários podem utilizar os dados gerados para coletar informações sobre o comportamento dos destinatários em

relação às notificações recebidas, bem como suas preferências. Isso permite que os usuários refinem seus *Workflows*, conteúdos de notificações e estruturem as preferências dos destinatários visando obter as melhores métricas de leitura e engajamento com as notificações. Portanto, o serviço de *Analytics* possui grande poder para auxiliar os usuários a reterem mais destinatários em seus produtos e, consequentemente, aumentar sua receita.

Figura 5.26: Visão geral do serviço de *Analytics* do sistema de notificações.



No estado atual do sistema de notificações construído, o serviço de *Analytics* fornece o histórico de eventos, mas ainda não realiza análises e inferências automáticas que possam auxiliar os usuários na tomada de decisão, cabendo a eles a análise dos dados. Entretanto, tomou-se o cuidado de que os eventos fossem armazenados utilizando formatos e tecnologias que facilitassem a implementação destas funcionalidades.

Para que o serviço de *Analytics* possua insumos para permitir a visualização do histórico e futura análise destes dados, é necessário que algumas informações geradas por outros serviços do sistema sejam consumidas ou replicadas. A visão geral de como o sistema de notificações implementa tais capacidades pode ser visualizada na Figura 5.26.

Assim, o serviço de *Analytics* pode ser entendido como um serviço composto por funções do AWS Lambda, as quais são invocadas através de requisições para a API REST exposta pelo API Gateway ou por meio de eventos emitidos por outros serviços que compõem o sistema de notificações. Para armazenar as informações recebidas por outros serviços do sistema e disponibilizadas através da API, optou-se por utilizar a solução do PlanetScale para o serviço de Vitess gerenciado, o que favorece o modelo *serverless* e garante escalabilidade de maneira totalmente gerenciada.

5.6.2 Entidades e Banco de Dados

Dado que o estado atual do serviço de *Analytics* é amplamente baseado nos dados e eventos emitidos por outros serviços do sistema de notificações, ele não introduz novas entidades ao sistema. Assim, as entidades presentes neste serviço foram originadas em outros serviços e replicadas para o serviço de *Analytics*, o qual utiliza esses dados com um novo propósito. Por isso, embora as entidades sejam as mesmas, a representação de algumas delas costuma ser simplificada. Assim, nesta Seção, elas são descritas brevemente, pois grande parte dos detalhes estão contidos na Seção 5.5.2.

A entidade de *Organization*, como no serviço de *Notification*, representa um usuário do sistema de notificações. Conforme descrito na Figura 5.6, esta entidade tem origem no serviço de IAM e é replicada assincronamente para este serviço. Logo, a representação desta entidade é idêntica àquela utilizada pelo serviço de *Notification*.

Da mesma forma, a entidade de *Workflow* possui o mesmo significado que a entidade de mesmo nome no serviço de *Notification*. Através de um processo de replicação de dados baseado em eventos, como `workflow.created`, o serviço de *Analytics* armazena referências aos *Workflows* construídos pelos usuários no serviço de *Notification*. Estas referências contêm apenas o identificador único da entidade original e seu nome. Elas são usadas para compor os dados do histórico de envio e futuros relatórios.

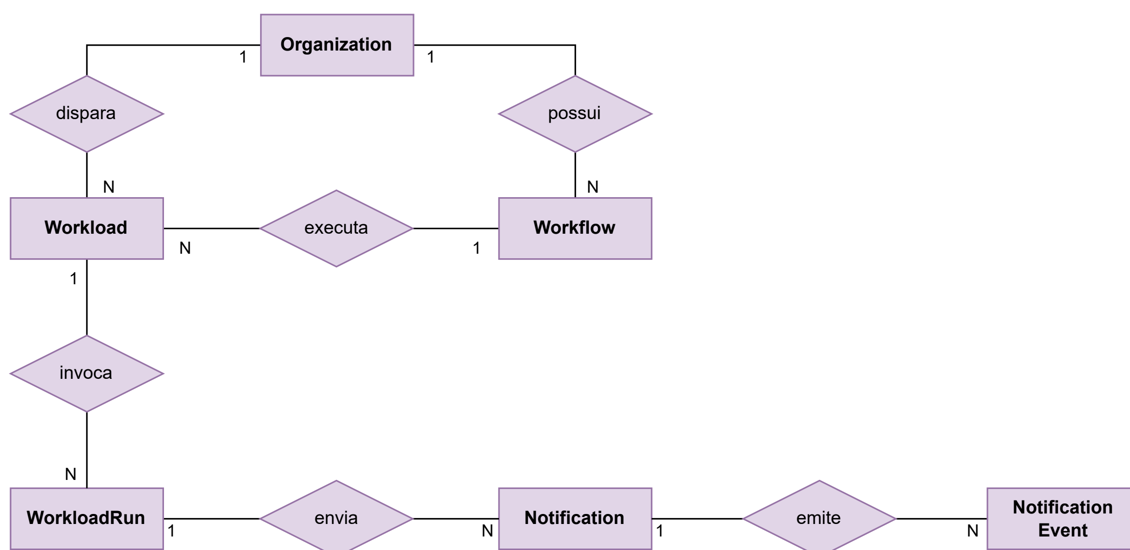
Os *Workloads* criados no serviço de *Notification* através de requisições de envio, também são replicadas para o serviço de *Analytics*. Igualmente, as entidades de *Workload Run* são replicadas para o serviço de *Analytics* após o processo de distribuição da carga de trabalho que ocorre no serviço de *Notification*. Estas entidades possuem o mesmo significado e representação em ambos serviços, embora sejam usadas para propósitos distintos.

Toda notificação enviada através do sistema de notificação é armazenada no serviço de *Anlytics* para que seja usada como insumo para análises e consultas do histórico de atividades. A representação utilizada é a mesma do serviço originador da entidade, o serviço de *Notification*.

Por fim, os eventos associados às notificações também são replicados para o serviço de *Analytics* para serem utilizados para compor históricos e análises. Neste serviço, a representação da entidade é feita da mesma maneira que no serviço de *Notification*, visto que todas as informações contidas nos eventos e notificações podem ser usadas para compreender o comportamento do sistema e dos destinatários.

Assim, dado que as entidades presentes no serviço de *Analytics* são originadas em outros serviços, o relacionamento entre elas é semelhante ao definido nos demais serviços. Na Figura 5.27 podemos visualizar o diagrama entidade relacionamento que define a modelagem utilizada para o banco de dados do serviço de *Analytics*. Diferentemente dos demais serviços, os dados são armazenados em instâncias de MySQL acessadas através da solução de Vitess disponibilizada pelo serviço do PlanetScale. Os atributos presentes nas entidades não foram representados no diagrama, porém já foram descritos em detalhes na Seção 5.5.2.

Figura 5.27: Diagrama de Entidade Relacionamento das entidades presentes no serviço de *Analytics* do sistema de notificações.

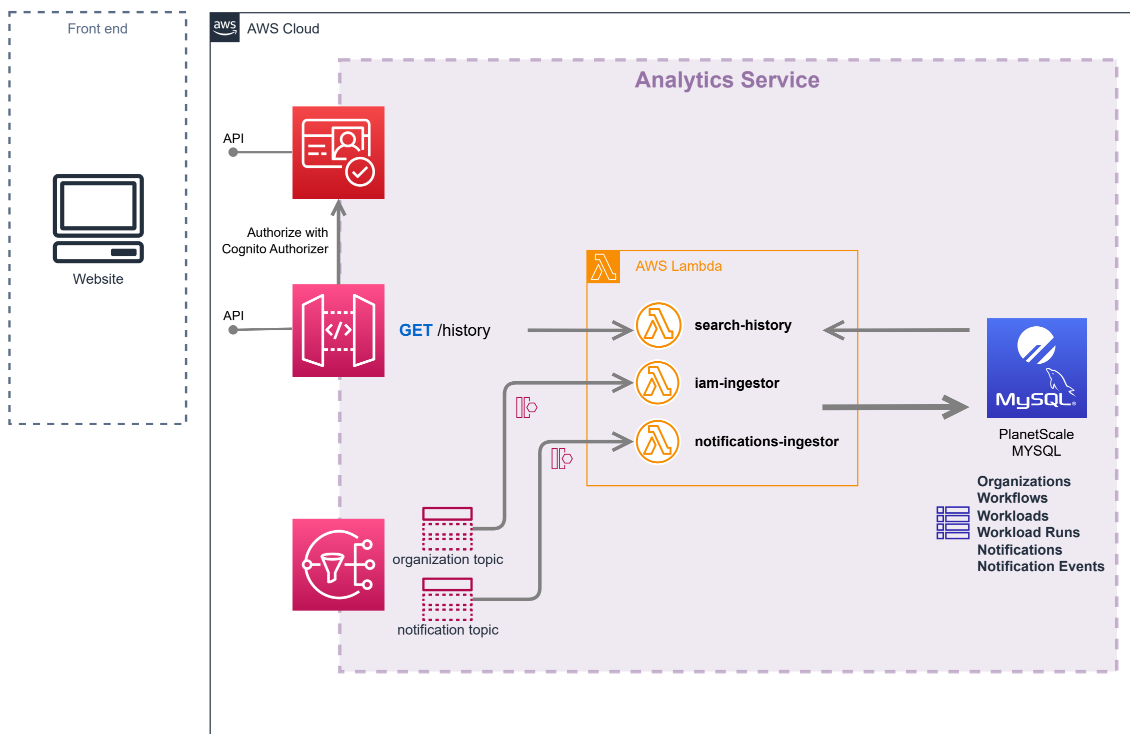


5.6.3 Arquitetura

Conforme comentado na Seção 5.6.1, o serviço de *Analytics* é baseado amplamente em um processo de replicação de dados originados nos outros serviços que compõem o sistema de notificações. A forma como os dados são replicados para o serviço de *Analytics* e disponibilizados para consulta através de uma API REST é descrita nesta Seção com o auxílio da representação arquitetural da solução contida na Figura 5.28.

Na parte inferior da Figura, podemos visualizar que uma das interfaces do serviço de *Analytics* é justamente o SNS. Por meio de tópicos do SNS, as mensagens e os eventos relevantes para o serviço de *Analytics* são consumidos e a replicação de dados ocorre. O serviço é inscrito em dois tópicos distintos: `organization` e `notification`.

Figura 5.28: Detalhes da arquitetura do serviço de *Analytics* do sistema de notificações.



O primeiro é utilizado para receber eventos relacionados às organizações registradas no serviço de IAM. Por exemplo, conforme descrito na Figura 5.6, o evento de `organization.created` é emitido pelo serviço de IAM quando uma nova organização é registrada no sistema de notificações. Na Figura 5.28, podemos verificar que ao receber este evento do tópico do `organization`, a função `iam-ingestor` é invocada através de uma fila SQS. Esta função é responsável por identificar o evento recebido e realizar a ação de replicação correta, por exemplo, inserir uma nova entidade de *Organization* no banco de dados ao receber o evento de `organization.created`.

Já o segundo tópico é utilizado para receber os eventos relacionados ao envio de notificações. Assim, toda ação relacionada ao envio de notificações é recebida e tratada pelo serviço, por exemplo, a definição de um novo *Workflow*, a criação de uma nova carga de trabalho e sua distribuição em partes menores e o envio de notificações para provedores. Portanto, alguns exemplos de eventos recebidos e tratados pelo serviço de *Analytics* são: `workflow.created`, `workload.created`, `workload_run.queued`, `notification.created`, `notification.queued` e `notification.undelivered`. Ao receber estes eventos, a função `notifications-ingestor` é invocada através de uma fila SQS e realiza o processo de identificação do evento e armazenamento no banco de dados MySQL.

Logo, o processo de replicação realizado no serviço de *Analytics* é baseado no consumo assíncrono de eventos relevantes através de tópicos do SNS e no armazenamento das entidades associadas no banco de dados MySQL. Após armazenar estes dados no banco de dados, eles ficam disponíveis para consulta através da API REST do serviço, bem como para futuras funcionalidades de análise e extração de conhecimento.

A API REST exposta por este serviço através do API Gateway é bastante simples e contém apenas uma rota: `GET /logs/history`. Esta rota é autorizada através do *Cognito Authorizer*, ou seja, requer que a API do Amazon Cognito seja utilizado no processo de autenticação para obtenção de *tokens* de acesso. Este método de autenticação e autorização foram escolhidos para esta rota, pois ela foi pensada para ser utilizada por uma aplicação *front-end*. Por fim, esta rota permite que sejam consultadas as notificações enviadas através de filtros, como data e hora, *Workflow* e *Workload*.

Assim, o serviço de *Analytics* pode ser entendido como um conjunto de funções do AWS Lambda, as quais são inscritas em tópicos relevantes do SNS para realizar a replicação de dados. Os dados replicados e armazenados são disponibilizados para consulta e análise através de rotas expostas pelo API Gateway.

5.6.4 Rotas da API

Nesta Seção, é descrita a única rota existente na API do serviço de *Analytics*, bem como os seus dados de entrada e resposta e método de autenticação. Os detalhes do funcionamento da rota presente neste serviço pode ser encontrada na documentação presente no Anexo B.

5.6.4.1 Histórico de Notificações

A rota `GET /logs/history` permite que o histórico de notificações enviadas e seus eventos possam ser consultados através da aplicação de alguns filtros. Visto que foi construída para ser utilizada por uma aplicação *front-end*, o método de autenticação aplicado nesta rota é baseado a integração com o Amazon Cognito conforme descrito na Figura 5.8. Além disso, através dos parâmetros de consulta da requisição HTTP (*query params*), é possível fornecer os campos descritos a seguir, os quais são usados para aplicar filtros aos dados consultados.

- `startDate`: parâmetro obrigatório que indica a data e hora inicial da janela de

tempo que deseja-se consultar o histórico de envios. A data deve estar no formato ISO 8601;

- `endDate`: parâmetro obrigatório que indica a data e hora final da janela de tempo que deseja-se consultar o histórico de envios. A data deve estar no formato ISO 8601;
- `workloadId`: parâmetro opcional que indica o identificador único do *workload* que enviou as notificações;
- `workflowId`: parâmetro opcional para indicar o identificador único do *workflow*, o qual foi executado para dar origem às notificações;
- `recipientId`: opcional e indica o identificador único do destinatário da notificação;
- `status`: opcional e indica uma lista de *status* que as notificações consultadas devem estar;
- `channels`: opcional e indica uma lista de canais usados para enviar as notificações;
- `pageAfter`: opcional e deve ser preenchido com o valor do cursor contido no campo `nextPage` da resposta desta rota para que sejam buscado os itens do histórico contidos na próxima página.

A partir da aplicação destes filtros, o histórico de envio pode ser consultado. A resposta da API para esta rota é feita utilizando uma estratégia de paginação de resultados baseada em cursores. Assim, nos casos de sucesso, a resposta tem o *status code 200 OK* e um objeto no formato JSON contendo dois campos: `items` e `nextPage`. O primeiro indica a lista de notificações presentes no histórico para os filtros aplicados. Já o segundo, se preenchido, indica o cursor que deve ser usado para buscar a próxima página de resultados, porém, nos casos em que não há mais resultados, o campo estará vazio. As notificações contidas na resposta possuem as informações sobre seu *status* atual, data e hora de criação, *workload* e *workflow*, bem como os detalhes de cada um dos eventos associados a cada notificação.

Caso ocorra algum erro de validação dos dados de entrada, a API retorna uma resposta com o *status code 400 Bad Request* e um objeto JSON com os detalhes do erro.

5.7 Integrações com provedores

Conforme descrito na Seção 2.2 e exemplificado na Seção 5.5, a integração com provedores de envio de notificações é uma funcionalidade fundamental de um sistema de notificações. Disponibilizar uma boa variedade de provedores e canais integrados ao sistema é um fator importante para conquistar novos usuários. Por isso, é indispensável que a adição de novos canais e provedores ao sistema seja uma tarefa simples. Neste trabalho, implementou-se a integração com três provedores de três canais distintos de maneira que a adição de novos provedores fosse simplificada.

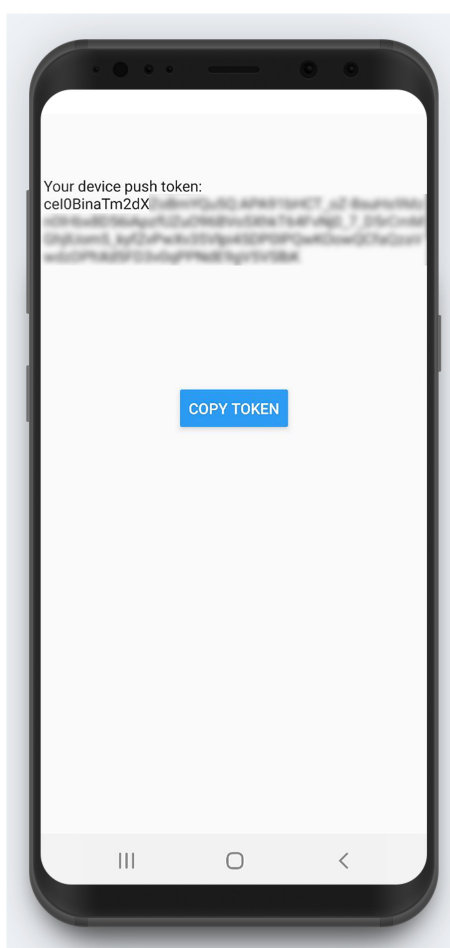
Para a lista de canais e provedores suportados, optou-se por escolher opções amplamente utilizadas: Twilio para SMS, SendGrid para e-mail e Firebase Cloud Messaging (FCM) para *push notification*. O processo de implementação para suportar um novo provedor de um canal já suportado pode ser dividido três etapas: (1) definir os novos dados necessários para integração com o provedor e adicionar suporte de cadastro na entidade *Integration* do serviço de *Notification*; (2) definir um novo filtro para o tópico associado aos eventos de `notification.queued` baseado no identificador do provedor; e (3) implementar uma nova função de *handler* no serviço de *Notification* para realizar a integração com a API do provedor. Já o processo de adição de um novo provedor para um canal ainda não suportado requer dois passo anteriores: (1) implementar a formatação de notificações usando o formato adequado para o canal; e (2) permitir que os dados necessários sejam adicionados ao cadastro de destinatários, por exemplo, *device token* para *push notifications*.

Logo, percebe-se que o suporte a novos provedores envolve basicamente etapas de adequação no cadastro da integração e implementação da integração com o provedor. Isso permite que novos provedores sejam adicionados sem a necessidade de modificar grandes porções de código, ou seja, o sistema de notificações implementado possui características de extensibilidade e manutenibilidade. Consequentemente, a tarefa de adicionar suporte a novos canais e provedores não envolve grandes esforços. Conforme a experiência percebida durante o desenvolvimento do trabalho, o esforço equivale a aproximadamente 6 horas de trabalho para ler a documentação do provedor e implementar os passos descritos anteriormente.

Para testar o envio de *push notification*, foi necessário implementar um aplicativo *Android* simples que permita recebimento de notificações. Isso foi feito com a utilização

da plataforma Expo[1] em conjunto com React Native. Na Figura 5.29 pode ser visualizada a tela do aplicativo desenvolvido, a qual apenas imprime o *device token* que autoriza o envio de notificações para o dispositivo e disponibiliza um botão para copiar o valor do *token*. Para os testes de envio de *push notifications* não foi necessário implementar outras funcionalidades no aplicativo. Embora o aplicativo tenha sido desenvolvido utilizando o React Native, o qual permite o desenvolvimento multiplataforma, o envio de *push notifications* através da integração com o FCM não foi testado em dispositivos *iOS* devido à falta de acesso a um dispositivo com este sistema operacional.

Figura 5.29: Tela do aplicativo desenvolvido para testar o envio de *push notifications*.



5.8 Análise comparativa com os trabalhos relacionados

Ao final do desenvolvimento e implementação do sistema de notificações descrito neste Capítulo, podemos comparar e analisar as funcionalidades e características deste sistema com aqueles descritos no Capítulo 4 sobre trabalhos relacionados. Na Tabela

[1]<https://expo.dev/> Acesso em 14 mar. 2023

5.1, é possível visualizar a tabela comparativa dos trabalhos relacionados com o sistema desenvolvido e, a seguir, os resultados da tabela serão discutidos.

Tabela 5.1: Análise comparativa das funcionalidades e características presentes neste trabalho e nas soluções de *Courier*, *Knock* e *SuprSend*.

Características	***Courier***	***Knock***	***SuprSend***	**Este Trabalho**
Abstrai canais e provedores	Sim	Sim	Sim	Sim
Suporta e-mail	Sim	Sim	Sim	Sim
Suporta SMS	Sim	Sim	Sim	Sim
Suporta *in-app*	Sim	Sim	Sim	Não
Suporta *chat*	Sim	Sim	Sim	Não
Suporta *push notifications*	Sim	Sim	Sim	Sim
Número total de provedores	48	23	13	3
Envio em *batch*	Não	Sim	Não	Não
Roteamento de notificações	Sim	Sim	Sim	Sim
Fluxos avançados de roteamento	Sim	Sim	Não	Sim
Gerenciamento de *templates*	Sim	Sim[2]	Sim	Sim
Editor de *templates* amigável	Sim	Não	Sim	Sim
Conteúdo abstraído em blocos	Sim	Não	Não	Sim
Preferências de destinatários	Sim	Sim	Não	Sim
Histórico de envio	Sim	Sim	Sim	Sim
Histórico detalhado e com erros	Sim	Sim	Não	Sim
Análise de performance	Não	Não	Sim	Não

Primeiramente, em relação ao suporte de canais e provedores, conforme discutido anteriormente, optou-se por dar suporte aos canais mais comuns (e-mail, *push notification* e SMS), bem como o principal provedor de cada canal. Consequentemente, este trabalho não possui suporte ao canal de notificações *in-app* e *chat* e também possui um número limitado de provedores. Dessa forma, foi possível alocar mais recursos e esforços na construção de outras funcionalidades importantes do sistema. Porém, tomou-se o cuidado de desenvolver o suporte aos canais e provedores de forma modular que facilitasse a adição de novas opções. Portanto, através da reutilização de código e de característica de extensibilidade, o sistema pode dar suporte a vários novos provedores, conforme descrito na Seção 5.7.

Em seguida, mecanismos de roteamento avançado foram adicionados através da construção de fluxos de envio personalizados usando *Workflows*, semelhantemente ao o que ocorre no *Courier* e *Knock*. Entretanto, devido à solução de agendamento descrita na Figura 5.24, é possível que a execução de um *Workflow* seja interrompida com precisão de segundos, o que não é garantido nos demais serviços. Por outro lado, a customização dos fluxos de envio ainda não possui suporte ao envio em *batch* como ocorre no *Knock*.

[2]O *Knock só permite gerenciamento de templates para e-mails.*

Além disso, embora o sistema desenvolvido não tenha focado na implementação de uma interface de usuário, ele possui suporte a *templates* para todos os canais disponíveis. Isto não ocorre no *Knock*, o qual possui uma série de limitações no sistema de *templating*, impedindo o gerenciamento para todos os canais e que pessoas sem conhecimento técnico construam seus próprios *templates*. Já o *SuprSend* possui suporte a *templates* para todos os canais, porém a construção de *templates* em cada canal ocorre de forma diferente, o que introduz constantes trocas de contexto durante a definição de *templates*. Isso também torna a curva de aprendizagem maior e impede o compartilhamento de definições de um *template* entre diferentes canais. Já o trabalho desenvolvido e o *Courier* utilizam blocos de conteúdo para abstrair textos, imagens, *links* e títulos, garantindo que a experiência de construção de *templates* seja uniforme e consistente para todos os canais, bem como permita a reutilização e compartilhamento de blocos de conteúdo entre canais.

O sistema desenvolvido também possui suporte ao gerenciamento de preferências de usuários com alto nível de granularidade por meio de seções e tópicos. Isso garante que as preferências possam ser gerenciadas de maneira simples pelos usuários do sistema e pelos destinatários de notificações. No *Courier* e no *Knock*, o suporte a gerenciamento está presentes, porém não garante, ao mesmo tempo, tanto poder de controle e usabilidade, em especial o *Knock*. Por fim, o *SuprSend* não possui nenhum mecanismo de controle.

Finalmente, os dados históricos são disponibilizados de maneira completa e detalhada, permitindo que os usuários do sistema realizem suas próprias análises de engajamento e performance, semelhante ao *Knock* e *Courier*. Entretanto, o sistema ainda não possui suporte a relatórios de performance automatizados como ocorre no *SuprSend*.

Através do conjunto de funcionalidades descritas, o sistema de notificações desenvolvido possui as funcionalidades essenciais para suprir as demandas dos usuários. Além disso, ele possui as características de extensibilidade necessárias para que funcionalidades sejam incrementadas e adicionadas.

6 DEMONSTRAÇÃO

Neste Capítulo, é demonstrado o funcionamento do sistema de notificações desenvolvido neste trabalho. Esta demonstração tem como foco as principais funcionalidades do sistema, ou seja, o envio de notificações, a criação e uso de *templates* e *workflows*, bem como o uso de preferências de destinatários. Sendo assim, para os testes demonstrados a seguir, é utilizada uma *Organization* previamente configurada com todos os provedores suportados e com chaves de acesso para API de *Notification* criadas. O teste simula o envio de notificações de boas-vindas para um novo membro de uma equipe *software*. Para isso, é utilizado um destinatário previamente cadastrado com o identificador `alencar`, o qual possui em seu perfil uma chave `role` com o valor `developer`. No Anexo A encontram-se detalhes sobre os dados das requisições usadas nesta demonstração.

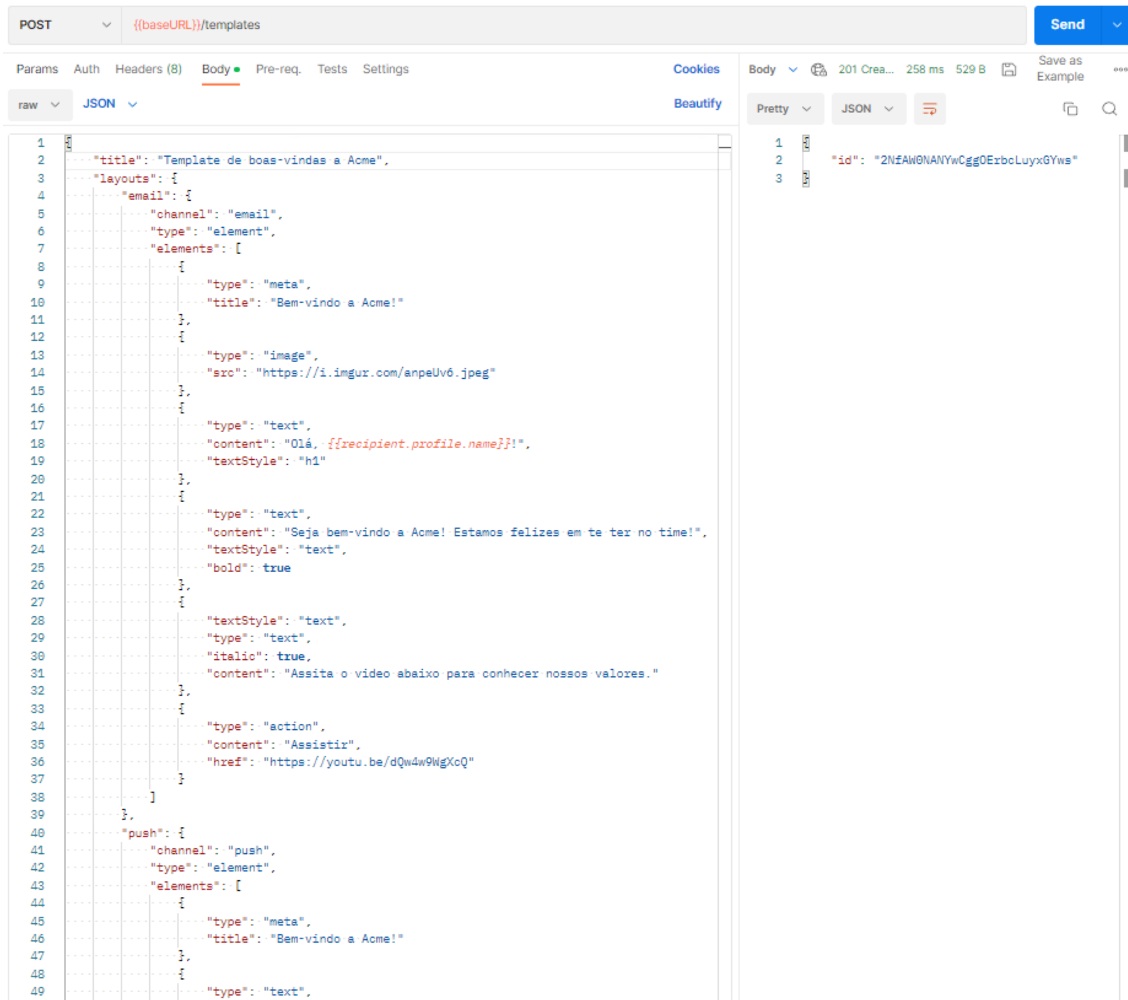
Dado que o escopo deste trabalho não contempla uma aplicação *front-end*, a interação com as rotas da API é realizada por meio de uma ferramenta chamada Postman[1]. Esta ferramenta permite a colaboração de desenvolvedores para projetarem, documentarem e testarem APIs através de uma interface de usuário, a qual possibilita a realização de requisições HTTP. Assim, quando uma requisição for realizada, uma captura de tela é apresentada demonstrando os resultados do teste.

6.1 Construção de um *template*

Para construir e definir um novo *template*, utilizamos a rota definida por `POST /templates` com a especificação dos blocos de conteúdo vinculados ao *template* para cada canal de interesse. No exemplo demonstrado na Figura 6.1, é definido um *template* de boas-vindas para novos membros de uma empresa usando e-mail e *push notifications*. Podemos visualizar que a API retornou o *status code 201 Created* no canto superior direito, bem como o identificador único do *template* criado. O corpo da requisição completo pode ser encontrado no Anexo A.

[1]<https://www.postman.com/> Acesso em 20 mar. 2023

Figura 6.1: Exemplo de requisição para criar *template* com definições para e-mail e *push notifications*.

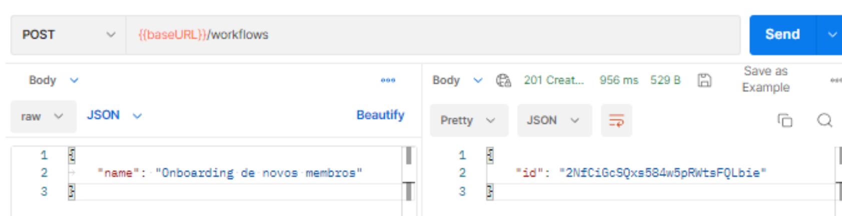


6.2 Construção de um *workflow*

Para construir e definir um novo *workflow*, utilizamos primeiramente a rota `POST /workflows` com o nome do *workflow*. Esta requisição pode ser visualizada na Figura 6.2, a qual permite a visualização do retorno da API, no canto superior direito, com o *status code 201 Created* e o identificador único do *workflow*. Na sequência, este identificador é utilizado com a rota `PUT /workflows/:id` para definir as etapas do *workflow*. Na Figura 6.3, podemos visualizar que a API retornou o *status code 200 OK* e o corpo da resposta vazio. O corpo destas requisições pode ser vistos em detalhes no Anexo A.

Neste *workflow*, foram utilizadas seis etapas, sendo cinco etapas de envio e uma etapa de *delay*. Inicialmente, o *workflow* realiza o envio de duas notificações de boas-vindas da empresa para o novo membro, uma é destinada para o e-mail e outra para o

Figura 6.2: Exemplo de requisição para criar novo *workflow*.



aplicativo Android através de *push notification*. Em seguida, realiza-se uma interrupção de cinco minutos na execução do *workflow*. Por fim, caso o novo membro seja do time de *marketing* (`role` com valor de `marketing`), um e-mail personalizado do time é enviado. Entretanto, se o membro for do time de desenvolvedores (`role` com valor de `developer`), é enviado um e-mail e um SMS com boas-vindas do time.

6.3 Execução do *workflow*

Para executar o *workflow* construído, podemos realizar uma requisição para a rota `POST /notifications` utilizando o tipo de envio `workflow` e o identificador do *workflow*. Nesta etapa da demonstração, utilizamos o destinatário cadastrado com o identificador `alencar`. Este destinatário foi cadastrado como pertencendo ao time de desenvolvedores (`role` com valor de `developer`). Logo, a etapa que realiza o envio de boas-vindas do time de *marketing* não deve ser executada. Além disso, foi criado uma seção e tópico de preferências para as notificações de boas-vindas. O destinatário utilizado nestes testes foi configurado para que deseje receber boas-vindas do time de desenvolvedores apenas através de SMS. Portanto, a notificação de boas-vindas do time através de e-mail não deve ser enviada para o destinatário.

Na Figura 6.4, podemos visualizar que a rota de envio foi utilizada e a API retornou a resposta com o *status code 202 Accepted* e um identificador do *workload*.

6.4 Visualização do histórico e notificações

Para conferir o histórico de eventos associados a execução do *workflow*, podemos utilizar a rota `GET /logs/history` fornecendo a data e hora inicial e final, bem como o identificador do *workload*. Na Figura 6.5 podemos visualizar a execução desta requisição. No lado direito da Figura, podemos verificar que, para este *workload*, foram geradas quatro notificações, o que é resultado da execução de quatro etapas de envio no *workflow*.

Figura 6.3: Exemplo de requisição para definir etapas de um *workflow*.

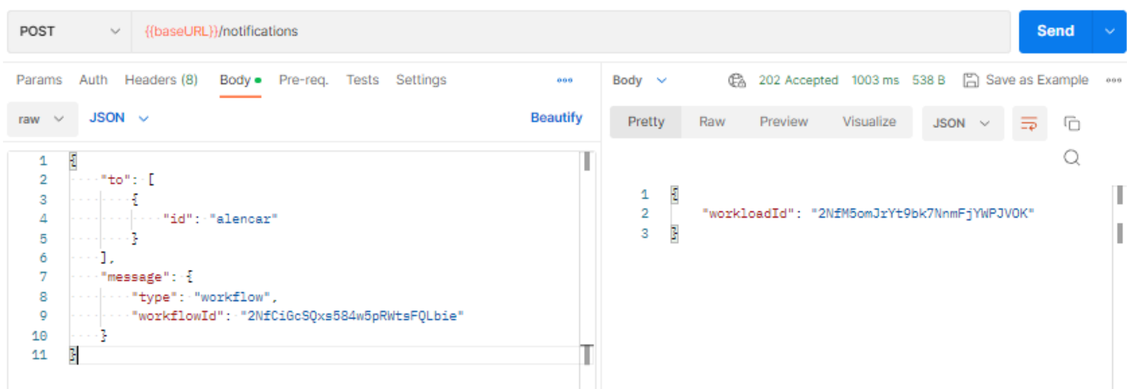

```
PUT {{baseURL}}/workflows/2NfCiGcSQxs584w5pRWtsFQLbie

{
    "name": "Onboarding de novos membros",
    "steps": [
        {
            "metadata": {
                "name": "Enviar email de boas-vindas"
            },
            "type": "send",
            "channel": "email",
            "message": {
                "type": "template",
                "templateId": "2NfAW0NANYwCggOErbcLuyxGYws",
                "flow": {...
                }
            }
        },
        {
            "metadata": {
                "name": "Enviar push de boas-vindas"
            },
            "type": "send",
            "channel": "push",
            "message": {
                "type": "template",
                "templateId": "2NfAW0NANYwCggOErbcLuyxGYws",
                "flow": {...
                }
            }
        },
        {
            "metadata": {
                "name": "Aguardar 5 minutos"
            },
            "type": "delay",
            "delay": {
                "type": "duration",
                "unit": "minutes",
                "value": 5
            }
        },
        {
            "metadata": {
                "name": "Enviar boas-vindas marketing"
            },
            "type": "send",
            "channel": "email",
            "message": {
                "type": "template",
                "templateId": "2NfC7X6qQ9FCyRFkNXZ0C7Tr5zF",
```


A etapa de envio das boas-vindas da equipe de *marketing* não foi executada devido às condições impostas na etapa. Por outro lado, as condições adicionadas nas etapas de envio de boas-vindas do time desenvolvedores foram satisfeitas e, por isso, estas notificações foram criadas. Entretanto, como podemos visualizar na resposta presente na Figura 6.5, a notificação de boas-vindas da equipe de desenvolvedores através de um e-mail não foi enviada e apresenta o *status* `undelivered`, pois as preferências do destinatário indicavam que ele não desejava receber este tipo de notificação através de e-mail. As demais notificações foram enviadas e o *status* de cada uma delas pode ser visualizado na Figura. Já a resposta completa com os eventos vinculados a cada notificação podem ser encontradas no Anexo A.

Finalmente, as notificações enviadas podem ser visualizadas nas Figuras 6.6 e 6.7. Na primeira, é possível visualizar o e-mail resultante da notificação de boas-vindas da empresa. Já na segunda Figura, o *smartphone* da esquerda exibe a *push notification* de

Figura 6.4: Exemplo de requisição de envio de notificações usando um *workflow*.



boas-vindas da empresa, enquanto o *smartphone* da direita exibe o SMS de boas-vindas do time de desenvolvedores.

Figura 6.5: Exemplo de requisição consultar o histórico de envio filtrando por *workload*.

GET {{baseURL}}/logs/history?startDate={{15minAgo}}&endDate={{now}}&workloadId=2NfM5omJrYt9bk7NnmFjYWPJVOK Send

Params

Query Params

	Key	Value
☐	pageAfter	<value of nextPage provide...
☑	startDate	{{15minAgo}}
☑	endDate	{{now}}
☑	workloadId	2NfM5omJrYt9bk7NnmFjY...
☐	workflowId	<workflowId>
☐	recipientId	<recipientId>
☐	status	<status>
☐	channels	<channels>
	Key	Value

Body 200 OK 624 ms 1.2 KB Save as Example

Pretty Raw Preview Visualize JSON


```
1   {
2       "items": [
3           {
4               "id": "2NfMhX9eseGeHscmMxQ3cSfC9rf",
5               "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
6               "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
7               "recipientId": "alencar",
8               "status": "undelivered",
9               "channel": "email",
10              "sentAt": "2023-03-29T00:46:43.000Z",
11              "events": [...
24              ]
25          },
26          {
27              "id": "2NfMhUAPsTf8uakGQZWpF0M1L17",
28              "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
29              "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
30              "recipientId": "alencar",
31              "status": "delivered",
32              "channel": "sms",
33              "sentAt": "2023-03-29T00:46:43.000Z",
34              "events": [...
53              ]
54          },
55          {
56              "id": "2NfM61TIut63iVfGcNujSwZVDjX",
57              "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
58              "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
59              "recipientId": "alencar",
60              "status": "sent",
61              "channel": "push",
62              "sentAt": "2023-03-29T00:41:44.000Z",
63              "events": [...
76              ]
77          },
78          {
79              "id": "2NfM5zxcN16BL3ReN2Kvi4js2nM",
80              "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
81              "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
82              "recipientId": "alencar",
83              "status": "seen",
84              "channel": "email",
85              "sentAt": "2023-03-29T00:41:44.000Z",
86              "events": [...
111             ]
112         }
113     ]
114 }
```

Figura 6.6: E-mail de boas-vindas da empresa enviado ao destinatário.

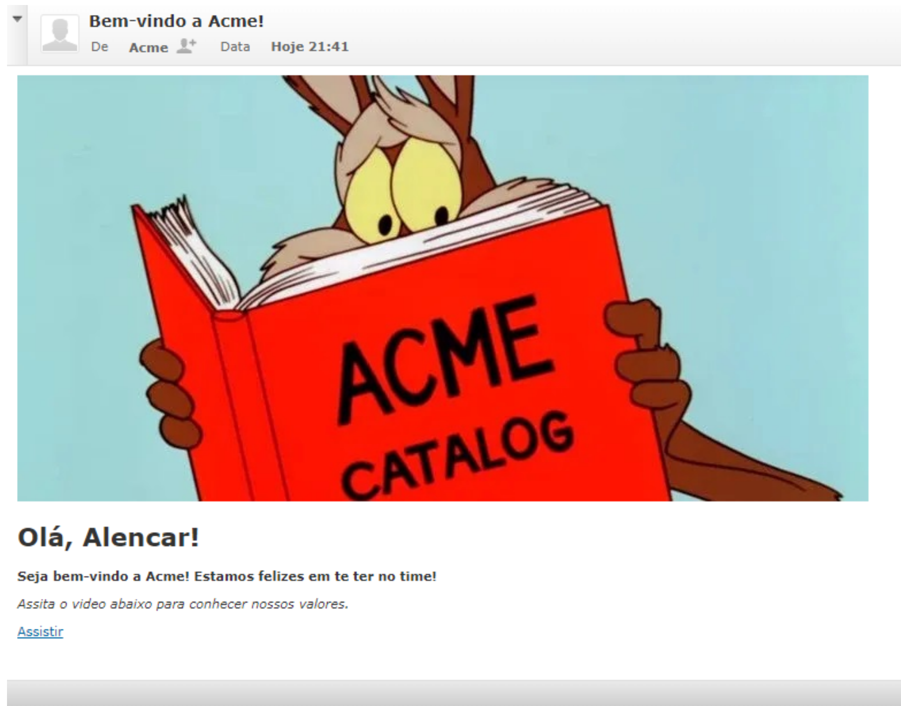
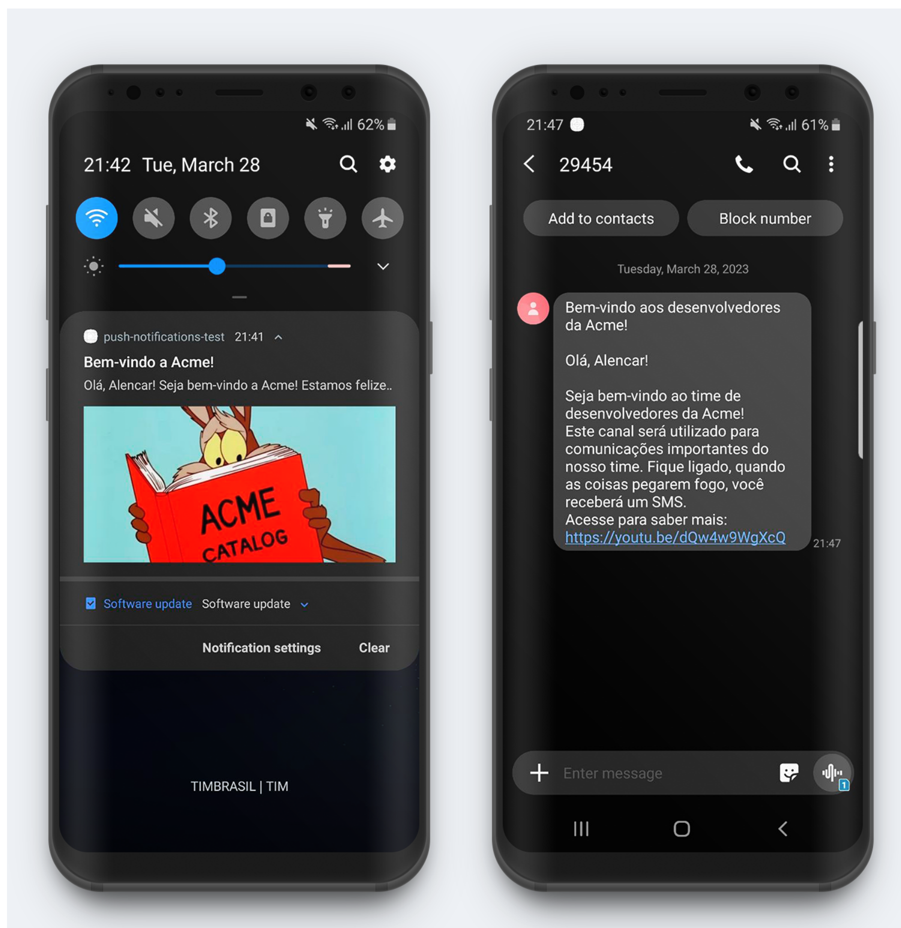


Figura 6.7: Na esquerda, a *push notification* de boas-vindas da empresa e, na direita, o SMS de boas-vindas do time de desenvolvedores.



7 EXPERIMENTOS E RESULTADOS

Neste Capítulo são descritos os experimentos realizados com sete usuários de diferentes níveis de conhecimento técnico com a finalidade de avaliar a usabilidade do sistema de notificações, em especial, da API de envio de notificações. Além disso, buscou-se avaliar o potencial do sistema e a qualidade da API em relação ao conjunto de boas práticas no desenvolvimento de APIs. Os resultados obtidos através dos testes com usuários e da análise da API desenvolvida são apresentados ao final do Capítulo.

7.1 Protocolo do Experimento

O experimento realizado com os usuários foi composto por três etapas. Na primeira etapa, os usuários foram convidados a responder a primeira seção do questionário, a qual coletava informações sobre o perfil do usuário através de questões objetivas, por exemplo, sua faixa etária, nível de escolaridade, experiências prévias com a utilização de APIs REST e experiências em relação ao envio de notificações. Na sequência, na segunda etapa do experimento, solicitou-se que os usuários realizassem um conjunto de tarefas utilizando a API REST e o sistema de notificações. Por fim, após a realização das tarefas, os usuários foram convidados a responderem a segunda parte do questionário, o qual avaliava sua experiência ao utilizar a API, bem como suas percepções em relação ao sistema de notificações e a qualidade da API utilizada. Nestas questões de avaliação, foi utilizado um conjunto de questões de escala linear de 1 a 5, em que "1" representa "Discordo totalmente" e "5" representa "Concordo totalmente". A aplicação do questionário foi realizada através da ferramenta Google Forms.

Em relação à realização das tarefas, os experimentos foram conduzidos de forma remota através da utilização de ferramentas de chamada de voz, como o Google Meet. Ao iniciar cada experimento, os usuários recebiam uma breve introdução sobre o sistema de notificações, sua API e seus objetivos. O acesso à documentação da API era fornecido neste momento para auxiliar na realização das tarefas. Além disso, os usuários possuíam acesso a uma coleção da ferramenta Postman[1] contendo exemplos de requisições presentes na documentação. A coleção utilizada, bem como as documentações fornecidas, podem ser encontradas no Anexo B. Esta ferramenta é amplamente difundida no mercado e por isto foi escolhida para auxiliar nos testes.

[1] <https://www.postman.com/> Acesso em 20 mar. 2023

Dado que objetivo principal do experimento era avaliar a usabilidade da API e suas funcionalidades, utilizou-se uma conta do sistema de notificações previamente configurada com as integrações com os provedores e, também, as chaves de acesso. Assim, o conjunto de tarefas aplicadas aos usuários pode conter mais tarefas relacionadas ao envio de notificações, ou seja, permitiu focar os testes no objetivo do experimento. Logo, para realizar as tarefas, os usuários utilizaram seus próprios computadores e ferramentas. Por fim, ao longo do desenvolvimento das tarefas, os usuários eram observados e acompanhados, de maneira que dúvidas pontuais fossem sanadas com o auxílio da documentação fornecida. A lista de tarefas solicitadas aos usuários encontra-se descrita a seguir.

1. Através da rota `PUT /recipients/:id`, cadastre-se como um destinatário do sistema de notificações. Ao perfil do destinatário (objeto `profile`), adicione um campo chamado `name` contendo seu nome e um campo `role` contendo a *string* `"admin"`. Para enviar notificações através de e-mail ou SMS é necessário fornecer valores válidos para os campos `phoneNumber` e `email`.
2. Utilize a rota `POST /notifications` para enviar uma notificação do tipo `element` contendo um elemento de cada tipo a seguir: `meta`, `text` e `action`. Para isso, o campo `type` deve ser preenchido com `"element"` e o *array* `elements` deve ser preenchido com os elementos que compõem sua mensagem. Adicione `{{recipient.profile.name}}` ao conteúdo de algum elemento para que ele seja preenchido com o nome do destinatário. O campo `flow.channel` deve ser preenchido com o canal que deseja enviar a notificação. A resposta da rota contém o `workloadId` associado a esta solicitação de envio.
3. Aguarde o recebimento da notificação e utilize a rota `GET /logs/history` para acompanhar o envio da notificação utilizando através de filtros no histórico de notificações. Utilize o valor de `workloadId` contido na resposta da requisição da tarefa anterior para filtrar por solicitação de envio.
4. Utilize os elementos construídos na tarefa 2 para criar um *template* para o canal desejado através da rota `POST /templates`. É obtido o identificador único do *template* como resposta.
5. Através da rota `POST /workflows`, crie um novo *workflow*. O identificador único do *workflow* é obtido como resposta.
6. Através da rota `PUT /workflows/:id`, adicione ao *workflow* criado no passo anterior as três etapas descritas a seguir:

1. Etapa de envio (`send`) usando o valor de `templateId` com o identificador único do *template* construído na tarefa 4.
2. Etapa de interrupção (`delay`) do *workflow* utilizando a estratégia do tipo `duration` e com duração de 1 minuto.
3. Etapa de envio (`send`) usando um *template* já cadastrado no sistema com o identificador a seguir: `2NQZd7p6AuDr4HDXtcIsMCjSJyP`. Adicione uma condição nesta etapa para que a notificação seja enviada apenas se o campo `recipient.profile.role` do destinatário conter o valor `“admin”`.

7. Utilize a rota `POST /notifications` para enviar uma notificação do tipo `workflow` usando o `workflowId` obtido na tarefa 5. Você deverá receber uma notificação logo em seguida e outra notificação destinada apenas para destinatários `"admin”` após 1 minuto. O envio pode ser acompanhado através da rota `GET /logs/history`, como realizado na tarefa 3.
8. *Opcional: utilizar as rotas de gerenciamento de preferências para definir seções e tópicos para os destinatários.*

Estas tarefas tem como objetivo apresentar aos usuários às principais funcionalidades do sistema de notificações. Por isso, o envio de notificações é realizado de duas formas diferentes e o usuário é instigado a utilizar os blocos de conteúdos definidos por *elements* para construir mensagens e *templates*. Da mesma forma, o processo de automação de fluxo de envio por meio de *workflows* é apresentado através da utilização de parte dos recursos possíveis. Além disso, após todo envio, os usuários são convidados a acompanharem os eventos de envio, recebimento e visualização atrelados às notificações enviadas. Através deste conjunto de tarefas, foi possível simular a utilização do sistema em tarefas cotidianas, como testar blocos de conteúdos antes de defini-los como um *template* reutilizável e criar *workflows* utilizando *templates*, etapas de interrupção e condições, bem como visualizar o histórico para compreender o funcionamento da integração em tarefas de *debugging* e o comportamento dos destinatários.

Este experimento foi conduzido com sete pessoas. Buscou-se convidar usuários que possivelmente seriam os responsáveis por interagir diretamente com o sistema de notificações através da API. Por isso, como demonstrado nos resultados do experimento, grande parte dos usuários são desenvolvedores com alguma experiência prévia no desenvolvimento de integrações com APIs, pois estes seriam capazes de avaliar a qualidade e usabilidade do sistema.

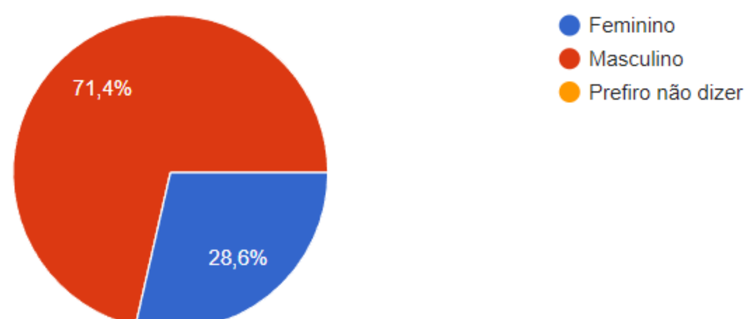
7.2 Resultados dos Experimentos

Através da aplicação dos questionários de perfil e avaliação do sistema de notificações com os sete participantes, o próprio Google Forms gerou uma série de gráficos com os resultados da pesquisa. Estes gráficos são apresentados a seguir juntamente com a discussão de seus dados. Vale ressaltar também que todos os sete usuários finalizaram todas as tarefas, ou seja, não houve desistências durante a aplicação dos testes. Assim, nesta Seção são apresentados os resultados para o questionário de perfil e avaliação do sistema.

7.2.1 Perfil dos usuários

A partir do gráfico contido na Figura 7.1, podemos visualizar que a maior parte dos usuários participantes identificam-se com o gênero masculino, mais especificadamente, houve a participação de cinco usuários homens e duas mulheres. Destes usuários, conforme a Figura 7.2, três estão na faixa de 20 a 24 anos, outros três na faixa de 25 a 29 anos e apenas um na faixa dos 30 a 34 anos. Já em relação à escolaridade, na Figura 7.3, apenas dois já concluíram o Ensino Superior, enquanto que o restante dos participantes são estudantes de Ensino Superior. Por fim, o gráfico da Figura 7.4 indica que a área de atuação dos participantes é majoritariamente o desenvolvimento *back-end*, contendo apenas um Líder Técnico e um desenvolvedor *front-end* entre os usuários participantes.

Figura 7.1: Gráfico dos resultados da pesquisa para questão 1.



Já em relação às experiências dos usuários, conforme a Figura 7.5, 100% dos participantes já desenvolveram alguma integração com uma API REST utilizando alguma

Figura 7.2: Gráfico dos resultados da pesquisa para questão 2.

2. Qual sua faixa etária?

7 respostas

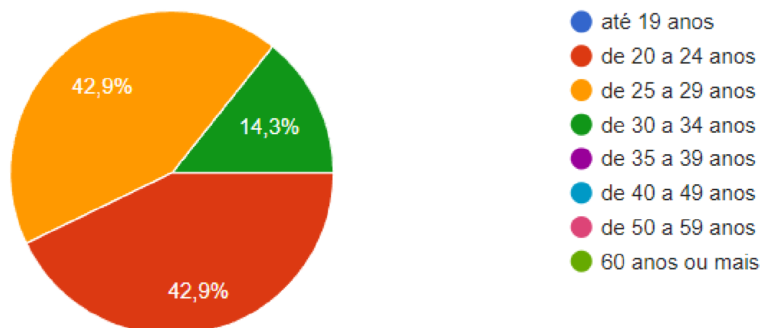
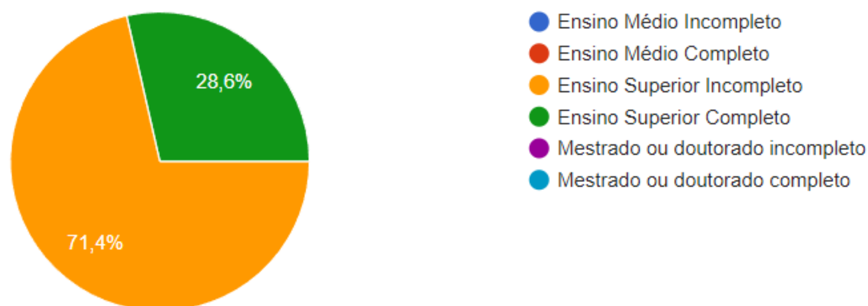


Figura 7.3: Gráfico dos resultados da pesquisa para questão 3.

3. Qual o seu nível de escolaridade?

7 respostas



linguagem de programação para realizar as requisições. Entretanto, no gráfico da Figura 7.6, podemos visualizar que, embora já tenham desenvolvidos tais integrações, a experiência dos usuários com estas tarefas não são totalmente iguais. Cinco dos participantes responderam que possuem boa experiência com estas integrações, enquanto dois participantes consideram sua experiência mediana ou baixa. O fato dos participantes possuir alguma experiência em desenvolver integrações com APIs REST significa que puderam utilizar tais experiências e conhecimentos para avaliar a API do sistema de notificações quanto a sua qualidade, usabilidade e adoção de boas práticas.

Por fim, nas questões 7 e 8 foi abordado o envio de notificações. Através dos resultados obtidos para estas questões e demonstrados nas Figuras 7.7 e 7.8, respectivamente, podemos perceber que todos eles já enviaram ou sentiram a necessidade de enviar notificações através de algum canal e provedor. Da mesma forma, todos os participantes dariam preferência para a utilização de um sistema de notificações com integrações com

Figura 7.4: Gráfico dos resultados da pesquisa para questão 4.

4. Qual a sua principal atuação no momento?

7 respostas

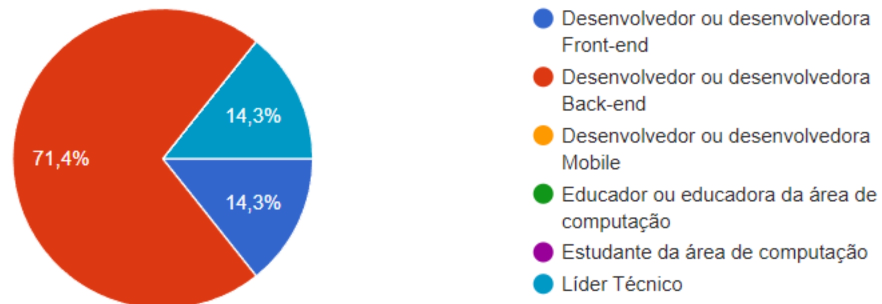
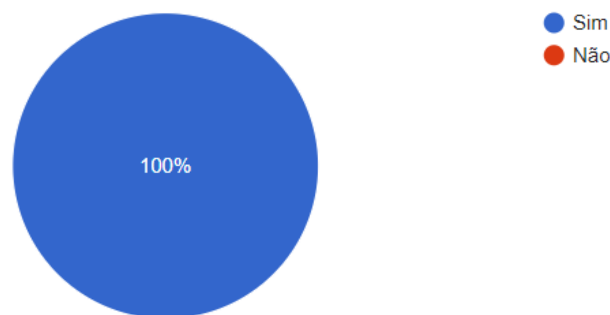


Figura 7.5: Gráfico dos resultados da pesquisa para questão 5.

5. Já desenvolveu alguma integração com uma API REST? Por exemplo, realizar requisições para uma API REST utilizando alguma linguagem de programação.

7 respostas



múltiplos provedores, ao invés de desenvolver suas próprias integrações. Isto pode servir como um indicativo de que a experiência de desenvolver integrações diretamente com os provedores não foi, ou não pareceu, simples o suficiente para que os participantes não enxergassem valor na utilização de um sistema de notificações como o desenvolvido neste trabalho.

7.2.2 Avaliação do sistema

Os resultados da questão 9, na Figura 7.9, demonstram que os usuários concordam com os possíveis ganhos de tempo que a adoção de um sistema de notificações pode proporcionar ao permitir que equipes *software* foquem no desenvolvimento de suas próprias soluções, ao invés de precisarem desenvolver sua própria infraestrutura para soluções secundárias como o envio de notificações.

Figura 7.6: Gráfico dos resultados da pesquisa para questão 6.

6. Caso já tenha desenvolvido, o quão experimente você é no desenvolvimento de integrações com APIs REST?

7 respostas

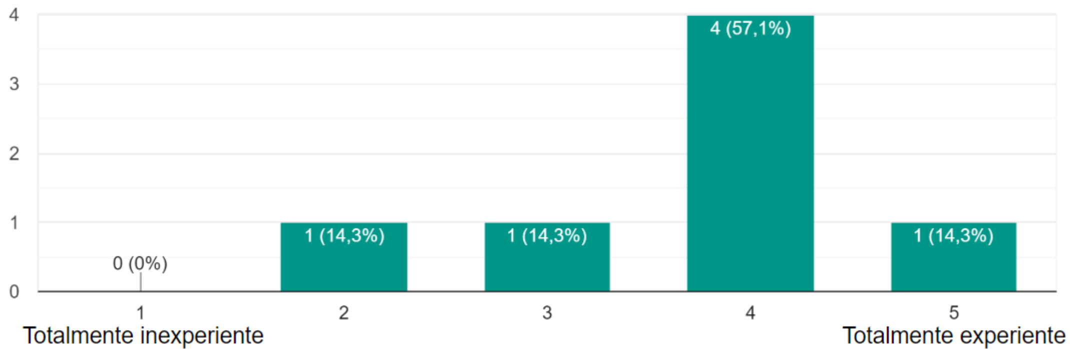
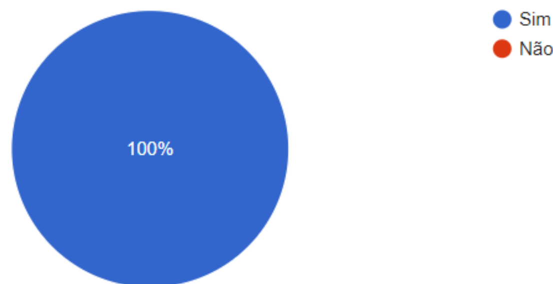


Figura 7.7: Gráfico dos resultados da pesquisa para questão 7.

7. Já utilizou ou sentiu a necessidade de utilizar um provedor para envio de algum tipo de notificação para destinatários? Por exemplo, SendGrid, Twilio, Mailgun, OneSignal, Firebase FCM, AWS SES, etc.?

7 respostas



Da mesma forma, os resultados da questão 10, na Figura 7.10, demonstram que os blocos de conteúdo, chamados de *elements*, proporcionam uma boa capacidade de abstração durante a construção de *templates*. Este recurso foi avaliado de forma positiva ao indicarem que usuários sem conhecimento técnico poderiam utilizar estes blocos para compor suas próprias notificações.

Assim, como os *templates* e sua forma de abstração foram bem avaliadas, outra funcionalidade importante do sistema de notificação também recebeu boas avaliações: o gerenciamento de preferências. Os resultados da questão 11, na Figura 7.11, demonstram que os recursos de gerenciamento de preferências disponibilizados pela API permitem que os *softwares* integrados ao sistema de notificações possam permitir que seus próprios usuários controlem as notificações que desejam receber de forma fácil e sem adição de complexidade no desenvolvimento.

Ao final da execução das tarefas, os participantes da pesquisa, dado os resultados

Figura 7.8: Gráfico dos resultados da pesquisa para questão 8.

8. Caso você precisasse implementar o envio de notificações através de múltiplos canais, o que você escolheria?
Sistema de notificações gerenciado: uma solução totalmente gerenciada com uma API única para todos canais e provedores de envio.
Solução própria: construir uma solução própria que realize a integração com os diferentes canais e provedores através de múltiplas APIs diferentes.

7 respostas

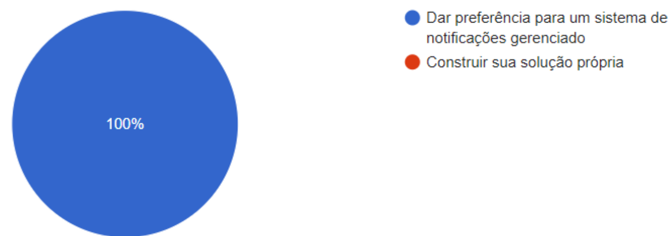
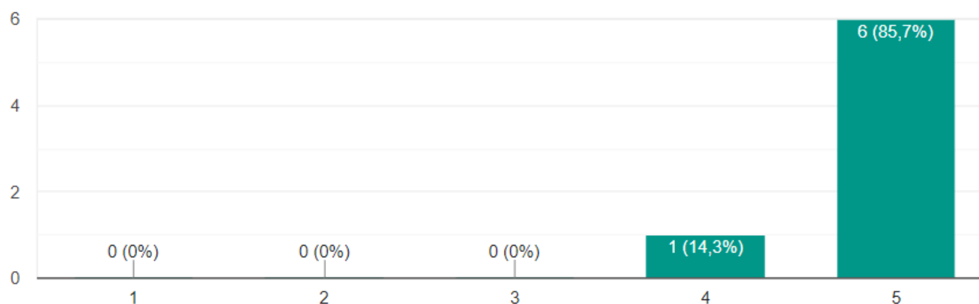


Figura 7.9: Gráfico dos resultados da pesquisa para questão 9.

9. As funcionalidades do sistema de notificações permitem que eu foque no desenvolvimento das regras de negócio do meu *software* sem precisar alocar muito tempo para implementar o envio de notificações e os recursos relacionados.

7 respostas



da questão 12 presentes na Figura 7.12, continuaram indicando que utilizariam um sistema de notificações, como o desenvolvido neste trabalho, ao invés de um construir sua própria solução. Isto reafirma que, mesmo após conhecer o sistema de notificações e sua API, as vantagens de utilizá-lo ainda são claras e mais favoráveis do que as demais.

As questões 13 e 14, as quais avaliavam a complexidade de enviar notificações através do sistema de notificações, demonstram que, de maneira geral, a tarefa de enviar notificações pode ser facilmente alcançada através do uso do sistema. Estes resultados podem ser visualizados na Figura 7.13 e 7.14.

O resultado apresentado na Figura 7.15 para a questão 15 reafirma que os participantes, de maneira unânime, identificam a velocidade no desenvolvimento proporcionada pela API implementada neste trabalho e a economia de tempo que ela pode proporcionar para equipes de *software*. Com certeza, isto é reflexo da simplicidade das abstrações da API.

Figura 7.10: Gráfico dos resultados da pesquisa para questão 10.

10. A composição de *templates* através dos blocos de conteúdo chamados de *elements* permite que usuários sem conhecimentos técnicos construam seus próprios *templates* através de uma UI sem auxílio de desenvolvedores.

7 respostas

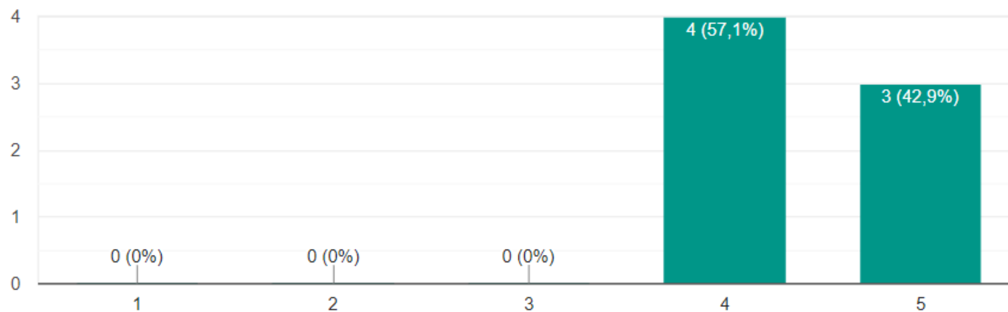
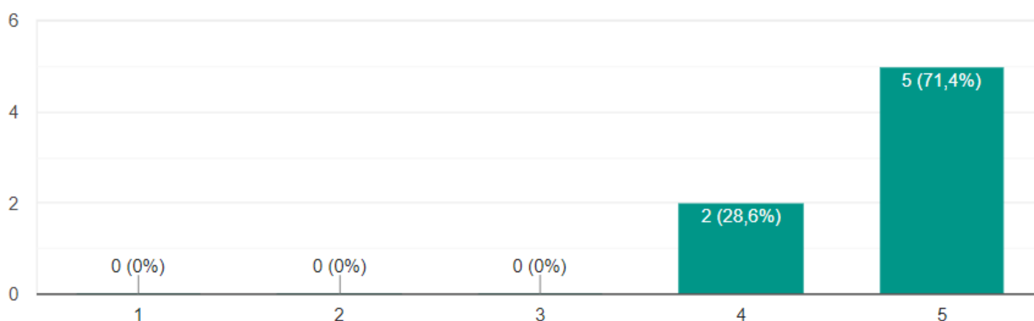


Figura 7.11: Gráfico dos resultados da pesquisa para questão 11.

11. Através das capacidades da API para gerenciar preferências de destinatários, eu posso, facilmente, dar poder aos meus usuários para decidirem quais conteúdos desejam receber.

7 respostas



Na Figura 7.16, os resultados da questão 16 também demonstram que, praticamente de forma unânime, o poder de automação dos *workflows* resulta em agilidade e economia de tempo dos desenvolvedores de *software* que necessitam enviar notificações através de fluxos personalizados, os quais costumam necessitar de desenvolvimento extra para serem possíveis.

As demais questões, avaliam a API do sistema de notificações em relação à adesão das boas práticas de desenvolvimento de APIs REST. Estas boas práticas são discutidas em maiores detalhes nas Seções seguintes. Entretanto, através dos resultados da Figura 7.17 da questão 17, pode-se notar que os participantes concordam que a API tenha consistência no formato dos dados das requisições e respostas no que diz respeito a estrutura dos objetos e nomeação de campos.

Da mesma forma, os resultados da questão 18, na Figura 7.18, demonstram que os caminhos das rotas da API foram nomeados de maneira consistente e intuitiva, o que faci-

Figura 7.12: Gráfico dos resultados da pesquisa para questão 12.

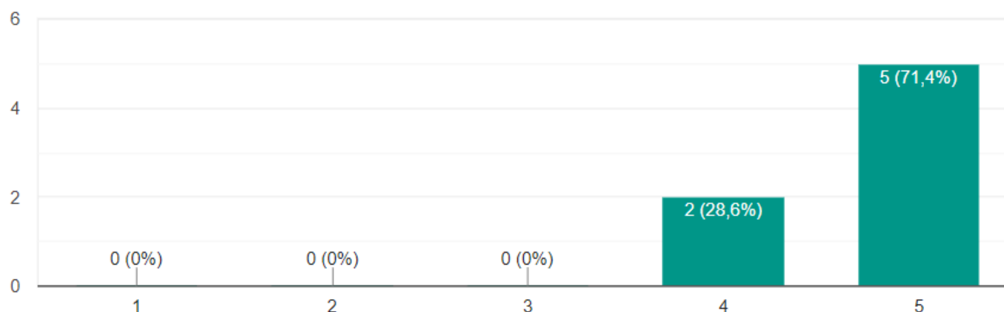
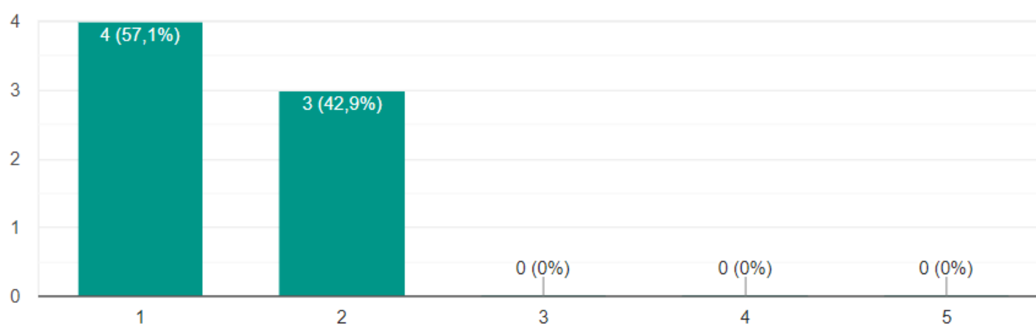


Figura 7.13: Gráfico dos resultados da pesquisa para questão 13.



lita a utilização da API e diminui a curva de aprendizagem. O mesmo pode ser visualizado na Figura 7.20 da questão 20, para a qual os participantes indicaram que a API implementada neste trabalho segue a melhores práticas de uma API REST. Essas características são discutidas em detalhes nas próximas Seções.

Entretanto, os resultados nas Figuras 7.19 e 7.21 das questões 19 e 20, respectivamente, indicam pontos de maior atenção, visto que há a presença de respostas medianas. Nestas questões, embora a maior parte dos usuários tenham concordado, houve um participante que se manteve neutro em relação a utilidades das mensagens de erro e na completude das documentações. Em especial, a questão 19 em que houve um empate de respostas "4" e "5", o que demonstra que talvez as mensagens de erro possam ser aprimoradas para conter mais informações e serem mais claras. Todavia, de maneira geral, o saldo nestas duas questões ainda é positivo e o comum foi concordar com a utilidade das mensagens de erro e clareza das documentações.

Figura 7.14: Gráfico dos resultados da pesquisa para questão 14.

14. É fácil enviar notificações através da API.

7 respostas

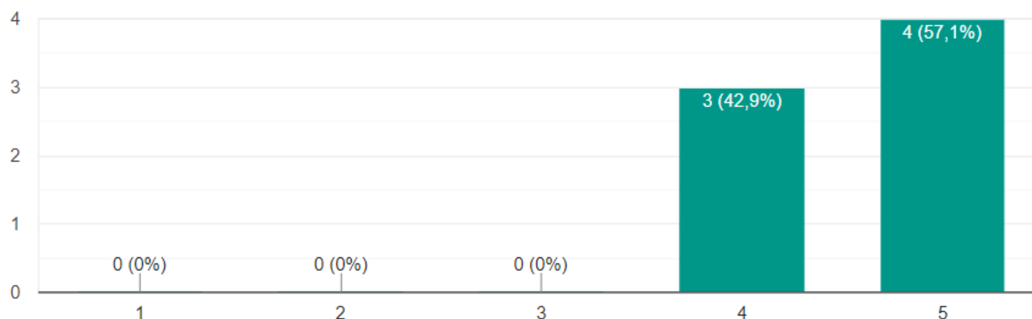
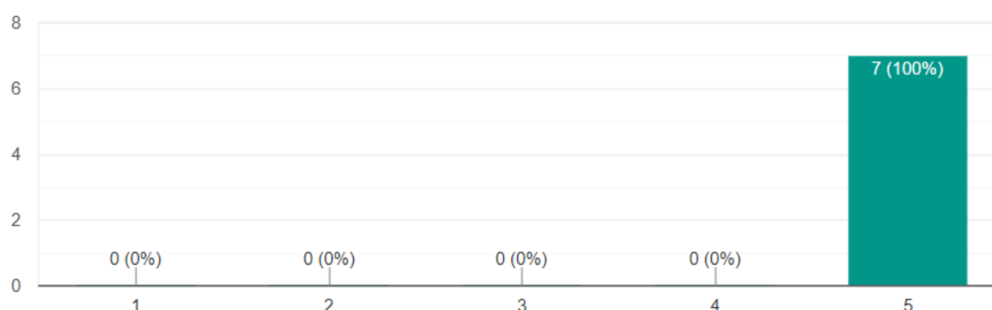


Figura 7.15: Gráfico dos resultados da pesquisa para questão 15.

15. Desenvolvedores conseguirão implementar integrações com a API de envio notificações (Notifications API) **rapidamente**. A integração neste caso não diz respeito a API voltada para o *front-end*.

7 respostas



Por fim, na questão 22 foi permitido que os participantes adicionassem comentários e sugestões para o trabalho. As respostas obtidas para esta questão estão transcritas a seguir.

"As funcionalidades foram bem implementadas e possuem bom fluxo de entendimento para o usuário." — PARTICIPANTE DESENVOLVER *back-end*

"A API e a documentação estão super claras, bem estruturadas e fazem total sentido com todo o contexto do sistema. Uma sugestão seria implementar uma rota para cancelar a execução de um workflow para algum destinatário específico." — PARTICIPANTE LÍDER TÉCNICO

"Muito legal a possibilidade de adicionar chaves extras aos destinatários. Isso permite que os desenvolvedores possam customizar o fluxo de notificações segundo a sua necessidade. Senti falta de condições de envio de mensagens baseado no status das mensagens anteriores enviadas em um mesmo workflow, por exemplo, para evitar que os

Figura 7.16: Gráfico dos resultados da pesquisa para questão 16.

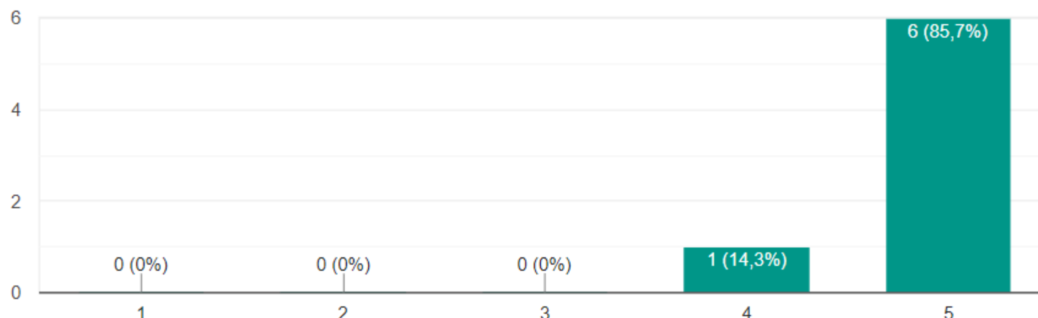
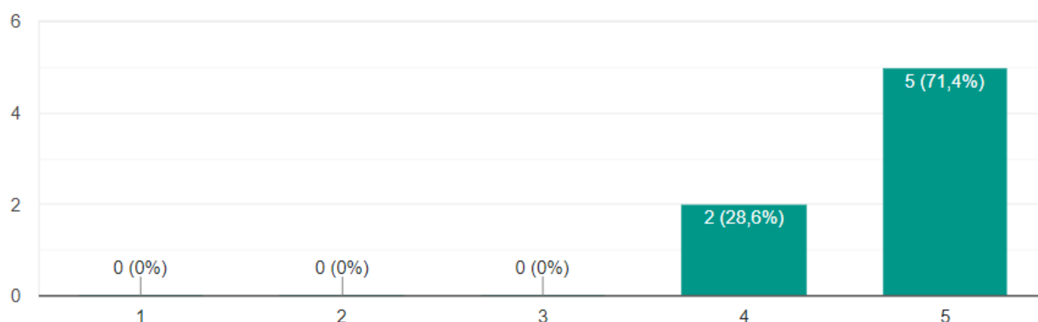


Figura 7.17: Gráfico dos resultados da pesquisa para questão 17.



usuários recebam a mesma comunicação várias vezes, caso o usuário já tenha aberto uma dada mensagem.“ — PARTICIPANTE DESENVOLVER *back-end*

Através destes comentários podemos verificar que, de fato, a API desenvolvida atendeu as principais necessidades dos usuários e o objetivo proposto por ela. Além disso, as sugestões de funcionalidades apontadas pelos participantes fazem total sentido ao sistema de notificações como um produto e podem ser implementadas facilmente para aprimorar as funcionalidades do sistema. A possibilidade de cancelar a execução de um *workflow* mediante requisição permitirá que a execução fluxos sejam interrompidas definitivamente quando esta tarefa deixar de fazer sentido para o *software* integrado ao sistema de notificações. Da mesma forma, a possibilidade de executar etapas de um *workflow* apenas nos casos em que uma notificação enviada por uma etapa anterior tenha, ou não, atingido determinado *status* é uma funcionalidade que aumenta ainda mais o poder das

Figura 7.18: Gráfico dos resultados da pesquisa para questão 18.

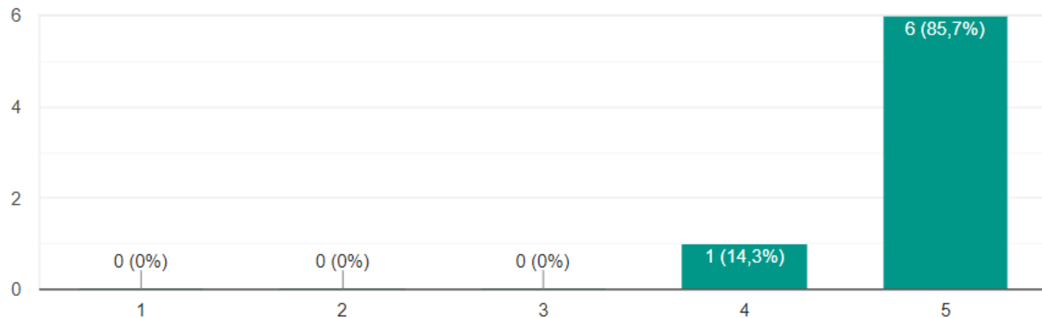
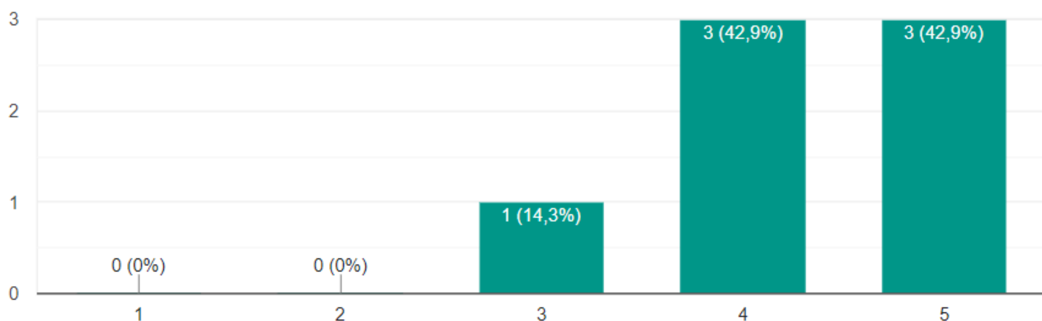


Figura 7.19: Gráfico dos resultados da pesquisa para questão 19.



automações dos *workflows*.

7.3 Análise de boas práticas

Após a implementação da API do sistema de notificações, podemos avaliar sua interface em relação às boas práticas no *design* de APIs REST. Assim, uma análise crítica pode ser realiza com base no que foi desenvolvido e nos resultados coletados nos experimentos com usuários. A seguir, descrevemos os principais padrões no desenvolvimento de APIs REST e discutiremos a adoção ou não dos mesmos no sistema de notificações.

Petrillo et al. (2016) categorizaram uma grande coleção de melhores práticas aplicáveis em diversos casos de uso. Uma das categorias definidas diz respeito às boas práticas aplicadas nas definições de *Universal Resource Identifier* (URI), ou seja, o caminho

Figura 7.20: Gráfico dos resultados da pesquisa para questão 20.

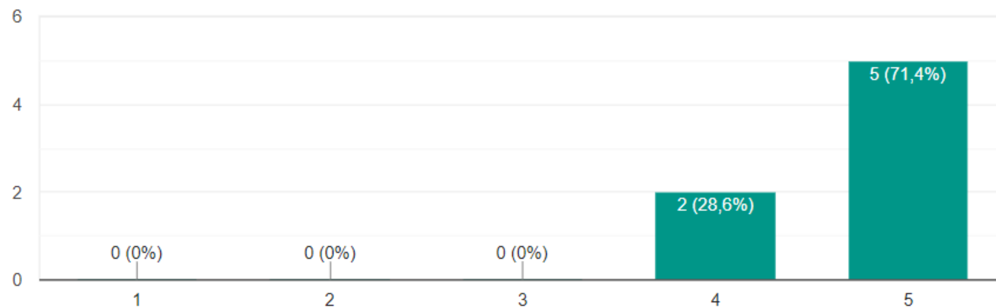
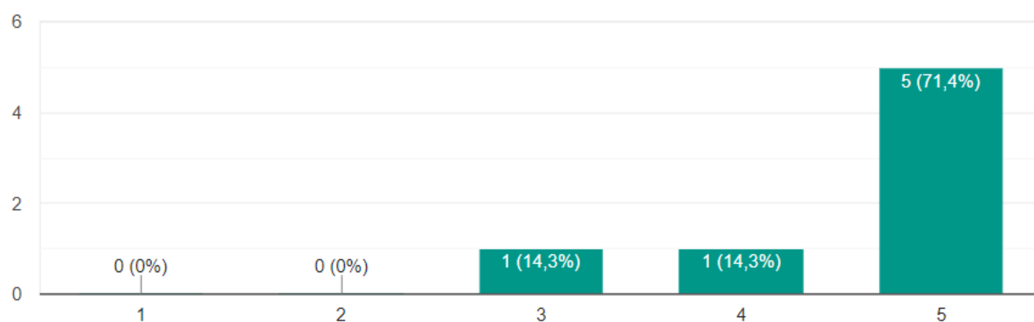


Figura 7.21: Gráfico dos resultados da pesquisa para questão 21.



das rotas da API usados para identificar os recursos. Em relação a esta categoria, os autores citam a utilização de barras (/) como separadores para identificar relações hierárquicas, bem como a preferência por utilizar substantivos no pural para nomear recursos da API e que, preferencialmente, não sejam utilizadas letras maiúsculas.

Ao analisar a API desenvolvida e descrita no Capítulo 5, verificamos estas boas práticas são adotadas, visto que todas rotas são definidas por letras minusculas e substantivos no plural, por exemplo, as rotas `POST /notifications` e `POST /templates`. Da mesma forma, barras (/) foram utilizadas para definir relações hierárquicas, como nas rotas `POST /preferences/sections` e `POST /preferences/topics`. Além disso, em Petrillo et al. (2016) ainda é citado que variáveis, em conjunto com barras (/), podem ser utilizados na URI para indicar a separação de hierarquias. Esse padrão é aplicado na rota `PUT /recipients/:recipientId/preferences` para indicar que se trata das

preferências de um determinado destinatário, conforme recomenda a boa prática.

A utilização de parâmetros de consulta na URI com a finalidade aplicar filtros e paginação aos resultados de uma requisição também é citado como uma boa prática por Petrillo et al. (2016) e Johnson (2019). Este padrão foi adotado na rota `GET /logs/history` do serviço de *Analytics* para permitir várias possibilidades de filtros e, também, a capacidade de paginar os resultados. Estes trabalhos também indicam que manter a consistência no padrão aplicado para a URI das rotas da API é uma boa prática para reduzir a curva de aprendizagem e momentos de frustração de usuários ao se depararem com inconsistências. Logo, ao avaliar o conjunto de rotas presentes na API do sistema de notificações, percebe-se que todas seguem o mesmo padrão e são bastante previsíveis. Essa característica também foi pontuada nos testes com os usuários e podem ser visualizadas nos resultados da questão 18 na Figura 7.18, a qual indica que os participantes concordam com a consistência e a adoção das boas práticas na definição de URI para as rotas.

Já em relação aos métodos aplicados nas requisições, os trabalhos de Petrillo et al. (2016) e Johnson (2019) descrevem como boa prática a utilização do método `POST` para criação de novos recursos, `PUT` para inserção e atualização de recursos, `GET` para consultar dados de recursos e, por fim, `DELETE` para deletar recursos. Novamente, as rotas presentes na API do sistema de notificações seguem este padrão de *design* e aplicam a semântica correta para cada método HTTP.

Em seguida, Petrillo et al. (2016) descrevem como boa prática no *design* de APIs a correta utilização da semântica de *status code* nas respostas retornadas aos usuários. O *status code 200 OK* deve ser utilizado para indicar sucessos não específicos e, isto pode ser visto em várias rotas de consulta, atualização e exclusão de recursos presentes na API. Já o *status code 201 Created* deve ser utilizado exclusivamente para indicar a criação de novos recursos e foi utilizado em todas as rotas com o método POST que indicam a criação de um novo recurso. Ainda para casos de sucesso, o *status code 201 Accepted* deve ser usado para indicar o início de uma tarefa assíncrona e foi utilizada na rota `POST /notifications` para indicar o processo assíncrono da solicitação de envio. Outros *status* de sucesso foram citados, como o *204 No content* para indicar resposta com o corpo intencionalmente vazio, porém, visto que seus casos de uso não foram aplicados, estes não são descritos.

Igualmente, os autores descrevem a semântica utilizada para os *status code* de resposta nos casos em que ocorrem falhas. *400 Bad Request* deve ser usado para designar

falhas não específicas, enquanto *404 Not Found* deve ser usado para designar recursos não mapeados pela API e *401 Unauthorized* quando ocorrem falhas autenticação da requisição. Conforme descrito no Capítulo 5, estes códigos de *status* também foram utilizados e, consequentemente, a API segue a boas práticas de para estes tipos de resposta.

No trabalho de Johnson (2019), o autor indica que a utilização de JSON como formato para os dados no corpo das requisições e respostas é um bom padrão de *design* a ser seguido. Evidentemente, é ressaltado que, ao adotar este formato, os campos presentes nestes dados devem seguir o mesmo padrão de nomenclatura para a consistência ser mantida. Da mesma forma, o formato dos valores deve ser consistente, por exemplo, datas serem representadas utilizando o mesmo formato em todos os dados de entrada e saída. A API do sistema de notificações, aplica estes padrões, visto que utiliza JSON como formato dos dados e todos os campos são nomeados usando o formato *camelCase*. Da mesma forma, o formato dos valores também é consistente, por exemplo, todo *enum* é representado por *strings* com letras minúsculas e toda data é representada no formato ISO 8601. A adesão a estes padrões também foi avaliada nas questões 17 e 20 do experimento com os usuários e os resultados obtidos na pesquisa indicam que os participantes concordam a consistência no formato dos dados, conforme as Figuras 7.17 e 7.20.

Entretanto, em relação aos detalhes de erros indicados no corpo das respostas, Johnson (2019) afirma que estes devem ser descritivos e claros o suficiente para que os usuários entendam o que causou o erro e corrigir os problemas presentes na requisição. Para isso, é sugerido que a resposta contenha um campo chamada `error` contendo informações sobre ele. Por exemplo, uma mensagem descrevendo o erro, um código definido pelo serviço que indique o erro em formato legível por humanos e, opcionalmente, um campo com detalhes extras sobre o erro. A API do sistema de notificações retorna erros em um formato semelhante, porém não adota a utilização do campo código conforme sugerido pelo autor. A adoção deste padrão pode trazer benefícios para os usuários da API, visto que facilitaria o tratamento de erros ao identificar o tipo dele através do código. As mensagens descrevendo os erros ainda poderiam ser mais claras e informativas, conforme os resultados da questão 19 demonstrado na Figura 7.19.

Por fim, os trabalhos de Petrillo et al. (2016) e Johnson (2019) citam a importância de APIs possuírem documentações públicas e completas para seus usuários utilizarem como guia durante o desenvolvimento de integrações. Na pesquisa aplicada aos participantes do teste de usabilidade, foi solicitada a avaliação da documentação da API, conforme os resultados questão 21 da Figura 7.19. Podemos perceber, que de maneira geral,

a documentação fornecida a eles era completa e clara. Entretanto, com certeza existem pontos de melhoria para aumentar a clareza através de mais exemplos e melhorias na estrutura.

7.4 Análise Geral dos Resultados

Ao final da aplicação do experimento e da análise de boas práticas a partir dos resultados da pesquisa e de embasamento teórico, nota-se que o sistema de notificações obteve bons resultados. Os potenciais usuários do sistema e participantes do experimento identificaram o sistema de notificações como uma melhor opção para o envio de notificações do que a integração direta com provedores. Os resultados indicam que isto foi motivado pela facilidade de envio através do sistema e das funcionalidades adicionais presentes nele, as quais permitem que eles foquem no desenvolvimento de suas soluções, ao invés de desenvolver funcionalidades complexas para envio de notificações. Consequentemente, eles indicam que isto resulta em maior agilidade de desenvolvimento e economia de tempo para disponibilizar essas funcionalidades em seus próprios *softwares*.

Durante a aplicação das tarefas, percebeu-se nitidamente a rápida compreensão da API por parte dos participantes. De maneira geral, as três primeiras tarefas eram suficientes para que eles se sentissem confiantes ao utilizar a API e sua documentação. Isso demonstrou que a curva de aprendizagem necessária para desenvolver uma integração com o sistema de notificações é curta e que a interface fornecida por ele é simples e intuitiva. Assim, as respostas dos participantes, as quais indicam a facilidade de enviar notificações e a consistência da interface, são reforçadas ao verificar a curva de aprendizagem durante a realização das tarefas.

Além disso, embora o trabalho não tenha focado em análises de custos financeiros atrelados a operação do sistema, as tecnologias utilizadas no trabalho possuem o modelo de cobrança *pay-as-you-go*, auxiliando na redução de custos. Visto que a operação de um sistema de notificações oscila entre intervalos de ampla utilização e intervalos de ociosidade, a adoção destas tecnologias garante que os custos nos momentos de ociosidades serão mínimos e que o sistema terá a escalabilidade necessária para os momentos de pico. Consequentemente, espera-se que a operação do sistema seja bastante eficiente em termos de custos financeiros, o que pode refletir na prática de preços mais acessíveis para usuários do sistema de notificações.

Por fim, ao analisar todos os resultados obtidos em conjunto, percebemos que o

sistema de notificações pode ser uma boa alternativa para a construção de uma solução própria, ao passo que aumenta a produtividade de equipes de *software*, as quais poderão focar em suas próprias regras de negócios e funcionalidades, ao invés de desenvolver sistemas paralelos para suportar funcionalidades secundarias. Logo, o sistema de notificações desenvolvido neste trabalho tem potencial para resolver o problema de seus usuários e se tornar um produto comercial.

8 CONCLUSÃO

Neste trabalho, visamos explorar as principais necessidades de aplicações e produtos de *software* no que diz respeito ao envio de notificações para seus usuários. A medida que o mercado de notificações cresce com a introdução de novos *software* e dispositivos, mais requisitos são exigidos durante o envio de notificações para suprir as exigências de usuários, companhias e instituições reguladoras. Assim, este trabalhou abordou a identificação e implementação das principais funcionalidades de um sistema de notificações que facilite o gerenciamento e desenvolvimento do envio de notificações.

Consequentemente, alcançamos o objetivo principal de construir de um sistema de notificações capaz de realizar envios através múltiplos canais e provedores a partir de uma única API REST, a qual abstrai as complexidades de cada canal. Para isso, implementamos a integração com múltiplos provedores e definimos um mecanismo de *templating* baseado em blocos de conteúdo. Objetivos complementares, como roteamento, gerenciamento de preferências e engajamento, também foram alcançados. O roteamento de notificações é personalizável através da construção de *workflows*, os quais definem fluxos automatizados de envio de notificações. Já para preferências, seções e tópicos podem ser vinculados a *templates* para que destinatários definam quais notificações desejam receber e através de qual canal. O engajamento dos destinatários é armazenado e disponibilizado para consulta para poder auxiliar em tomadas de decisões. Por fim, para o sistema ser resiliente aos picos de utilização, a escolha das tecnologias e da arquitetura foi baseada em capacidades de escalonamento automático, processamento assíncrono e que evite perdas de notificações.

Com isso, foi possível validar o sistema como uma solução para os problemas associados às complexidades do envio de notificações. Por meio de testes de usabilidade com usuários, identificamos que o sistema de notificações construído neste trabalho tem grande potencial de ser uma alternativa à manutenção de soluções próprias para este problema. Os participantes dos testes apontaram o sistema de notificações como sendo de baixa complexidade, intuitivo e de rápida integração. Para eles, isso representa maior agilidade no desenvolvimento de suas aplicações e economia de tempo, o que é um reflexo da API adotar um conjunto de boas práticas e fornecer as abstrações corretas. Por isso, afirmaram que preferem utilizar o sistema de notificações a implementar suas próprias soluções.

Entretanto, como limitações deste trabalho, destacamos as dificuldades de avalia-

ção da API REST do sistema de notificações construído. Dada a necessidade de usuários interagirem diretamente com a API para poderem avaliá-la, o perfil dos participantes acabou limitando-se àqueles com determinados conhecimentos técnicos, bem como refletiu em uma maior duração dos testes. Estas características impediram a realização de testes de usabilidade com um grande número de participantes.

Logo, trabalhos futuros podem focar na exploração de outros métodos de avaliação como forma de validação do sistema de notificações, bem como na análise dos requisitos tecnológicos para o sistema ser ofertado como um modelo de serviço. Além disso, a implementação de uma interface de usuário por meio de uma aplicação *front-end*, será fundamental para difundir o sistema construído e alcançar novos usuários. Ao mesmo tempo, a implementação das funcionalidades sugeridas durante os testes de usabilidade são importantes para a completude do sistema e podem ser exploradas em trabalhos futuros, bem como a adição de novos blocos de conteúdo para o sistema de *templating*, novas etapas de automação para *workflows* e a extração de conhecimento a partir dos dados de engajamento de destinatários. Por fim, a implementação do suporte a mais opções de canais e provedores é imprescindível para o sucesso da aplicação. Assim, a evolução do sistema de notificações pode significar a sua introdução no mercado como um competidor dos trabalhos relacionados.

Por fim, esperamos ainda que este trabalho possa ser utilizado como base ou referência técnica para aqueles que buscam desenvolver suas próprias soluções para o envio de notificações através de múltiplos canais e provedores, bem como as funcionalidades vinculadas a este sistema.

REFERÊNCIAS

AIRSHIP. **Push notifications & mobile engagement: 2021 Benchmarks**. 2021. Disponível em: <https://grow.urbanairship.com/rs/313-QPJ-195/images/Push_Notifications_Mobile_Engagement_2021_Benchmarks.pdf>. Acesso em: 11 de set. de 2022.

AMAZON. **What is a RESTful API?** 2022. Disponível em: <https://aws.amazon.com/what-is/restful-api/>. Acesso em: 04 de mar. de 2023.

AMAZON. **Amazon API Gateway Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/apigateway/index.html>. Acesso em: 05 de mar. de 2023.

AMAZON. **Amazon Cognito Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/cognito/index.html>. Acesso em: 05 de mar. de 2023.

AMAZON. **Amazon DynamoDB Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/dynamodb/index.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **Amazon EventBridge Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/eventbridge/index.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **Amazon Simple Notification Service endpoints and quotas**. 2023. Disponível em: <https://docs.aws.amazon.com/general/latest/gr/sns.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **Amazon Simple Queue Service Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/sqs/index.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **Amazon Simple Queue Service Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/sns/index.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **AWS CloudFormation Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/cloudformation/index.html>. Acesso em: 05 de mar. de 2023.

AMAZON. **AWS Lambda Documentation**. 2023. Disponível em: <https://docs.aws.amazon.com/lambda/index.html>. Acesso em: 04 de mar. de 2023.

AMAZON. **Overview of Amazon Web Services**. [S.l.], 2023. Disponível em: <https://docs.aws.amazon.com/pdfs/whitepapers/latest/aws-overview/aws-overview.pdf>. Acesso em: 04 de mar. de 2023.

ASLANPOUR, M. S. et al. Serverless edge computing: Vision and challenges. In: **2021 Australasian Computer Science Week Multiconference**. New York, NY, USA: Association for Computing Machinery, 2021. (ACSW '21). ISBN 9781450389563.

BAIRD, A. et al. **Serverless Architectures with AWS Lambda: Overview and Best Practices**. [S.l.], 2017. Disponível em: <https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf>. Acesso em: 04 de mar. de 2023.

BITTENCOURT, M. **Implementing a Transactional Outbox Pattern with DynamoDB Streams to Avoid 2-phase Commits**. 2020. Disponível em: <https://medium.com/ssense-tech/ed0f91e69e9>. Acesso em: 15 de mar. de 2023.

CARNEY, S. **The Developer's Guide to Building Notification Systems: Scalability and Reliability**. 2021. Disponível em: <https://www.courier.com/blog/scalability-and-reliability/>. Acesso em: 23 de set. de 2022.

DEBRIE, A. **AWS re:Invent 2019: I didn't know Amazon API Gateway did that**. 2019. Disponível em: <https://www.youtube.com/watch?v=yfJZc3sJZ8E>. Acesso em: 04 de mar. de 2023.

DEBRIE, A. **The DynamoDB Book**. 1. ed. [S.l.: s.n.], 2020.

DEBRIE, A. **AWS re:Invent 2021 - Data modeling with Amazon DynamoDB**. 2021. Disponível em: <https://www.youtube.com/watch?v=yNOVamgIXGQ>. Acesso em: 04 de mar. de 2023.

DIJK, L. V. **One million connections**. 2022. Disponível em: <https://planetscale.com/blog/one-million-connections>. Acesso em: 05 de mar. de 2023.

ELHEMALI, M. et al. **Amazon DynamoDB: A scalable, predictably performant, and fully managed NoSQL database service**. In: **USENIX ATC 2022**. [S.l.: s.n.], 2022. p. 1037–1048. ISBN 978-1-939133-29-8.

FIELDING, R. T. **Architectural Styles and the Design of Network-based Software Architectures**. Thesis (Doctoral dissertation) — University of California, Irvine, 2000.

GOODE, T. **The Developer's Guide to Building Notification Systems: Routing and Preferences**. 2021. Disponível em: <https://www.courier.com/blog/routing-and-preferences/>. Acesso em: 23 de set. de 2022.

GOODE, T. **The Developer's Guide to Building Notification Systems: User Requirements**. 2021. Disponível em: <https://www.courier.com/blog/the-developers-guide-user-requirements/>. Acesso em: 23 de set. de 2022.

GOODE, T. **How to Improve or Rebuild a Product Notification System: Video**. 2021. Disponível em: <https://www.courier.com/blog/how-to-improve-or-rebuild-a-product-notification-system-video/>. Acesso em: 23 de set. de 2022.

GUPTA, S. **Build vs Buy: The "to be or not to be" of Tech**. 2022. Disponível em: <https://www.courier.com/blog/build-vs-buy/>. Acesso em: 23 de set. de 2022.

HASAN, M. **State of IOT 2022: Number of connected IOT devices growing 18% to 14.4 billion globally**. 2022. Disponível em: <https://iot-analytics.com/number-connected-iot-devices>. Acesso em: 23 de ago. de 2022.

HASSAN, H. B.; BARAKAT, S. A.; SARHAN, Q. I. **Survey on Serverless Computing**. Hindawi Limited, London, GBR, v. 10, n. 1, jul 2021. ISSN 2192-113X.

HEINISCH, J. S. et al. **Investigating the Effects of Mood & Usage Behaviour on Notification Response Time**. [S.l.]: arXiv, 2022.

JOHNSON, T. **API design and usability**. 2019. Disponível em: <https://idratherbewriting.com/learnapidoc/evaluating-api-design.html>. Acesso em: 25 de mar. de 2023.

MANSOOR, M. et al. **Introduction to DevOps on AWS**. [S.l.], 2020. Disponível em: <https://docs.aws.amazon.com/pdfs/whitepapers/latest/introduction-devops-aws/introduction-devops-aws.pdf>. Acesso em: 04 de mar. de 2023.

MOORHEAD, S. **A year of 1 trillion emails: The customers who sent them: Twilio sendgrid**. 2022. Disponível em: <https://sendgrid.com/blog/1-trillion-emails-customers-who-sent-them/>. Acesso em: 11 de set. de 2022.

MORRISON, B. **Vitess for the rest of us**. 2022. Disponível em: <https://planetscale.com/blog/vitess-for-the-rest-of-us>. Acesso em: 05 de mar. de 2023.

NODE.JS. **Documentation**. 2022. Disponível em: <https://nodejs.org/en/docs/>. Acesso em: 05 de mar. de 2023.

NODE.JS. **The V8 JavaScript Engine**. 2022. Disponível em: <https://nodejs.dev/en/learn/the-v8-javascript-engine/>. Acesso em: 05 de mar. de 2023.

PEMBERTON, C. **Tap into the marketing power of SMS**. 2016. Disponível em: <https://www.gartner.com/en/marketing/insights/articles/tap-into-the-marketing-power-of-sms>. Acesso em: 23 de ago. de 2022.

PETRILLO, F. et al. Are rest apis for cloud computing well-designed? an exploratory study. In: SHENG, Q. Z. et al. (Ed.). **Service-Oriented Computing**. Cham: Springer International Publishing, 2016. p. 157–170. ISBN 978-3-319-46295-0.

PIELOT, M.; CHURCH, K.; OLIVEIRA, R. de. An in-situ study of mobile phone notifications. In: **Proceedings of the 16th International Conference on Human-Computer Interaction with Mobile Devices & Services**. New York, NY, USA: Association for Computing Machinery, 2014. (MobileHCI ’14), p. 233–242. ISBN 9781450330046.

PLANETSCALE. **PlanetScale documentation**. 2023. Disponível em: <https://planetscale.com/docs>. Acesso em: 05 de mar. de 2023.

REDHAT. **What is a REST API?** 2020. Disponível em: <https://www.redhat.com/en/topics/api/what-is-a-rest-api>. Acesso em: 04 de mar. de 2023.

REDHAT. **What is Infrastructure as Code (IaC)?** 2022. Disponível em: <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>. Acesso em: 04 de mar. de 2023.

SEELY, S. **Build vs buy: a guide for notification systems**. 2022. Disponível em: <https://knock.app/blog/build-v-buy-notifications>. Acesso em: 23 de set. de 2022.

SERVERLESS-FRAMEWORK. **Serverless Framework Documentation (V2)**. 2023. Disponível em: <https://www.serverless.com/framework/docs>. Acesso em: 04 de mar. de 2023.

SHAFIEI, H.; KHONSARI, A. Serverless computing: Opportunities and challenges. **CoRR**, abs/1911.01296, 2019.

TWILIO. **Investor relations**. 2022. Disponível em: <https://investors.twilio.com/>. Acesso em: 20 de ago. de 2022.

VISURI, A. et al. Understanding smartphone notifications' user interactions and content importance. **International Journal of Human-Computer Studies**, v. 128, p. 72–85, 2019. ISSN 1071-5819.

VITESS. **Documentation**. 2023. Disponível em: <https://vitess.io/docs/>. Acesso em: 05 de mar. de 2023.

VITESS. **History: Born at YouTube, released as Open Source**. 2023. Disponível em: <https://vitess.io/docs/16.0/overview/history/>. Acesso em: 05 de mar. de 2023.

WIGGERS, K. **Courier lands $35m to build a service for App Notifications**. 2022. Disponível em: <https://techcrunch.com/2022/06/22/courier-lands-35m-to-build-a-service-for-app-notifications/>. Acesso em: 20 de set. de 2022.

ZAKOWICZ, G. **Email, SMS, and push marketing statistics for ecommerce in 2022**. 2022. Disponível em: <https://www.omnisend.com/blog/email-sms-push-marketing-statistics-ecommerce/>. Acesso em: 20 de ago. de 2022.

ANEXO A — REQUISIÇÕES DO EXPERIMENTO

A seguir, encontram-se os principais dados das requisições utilizadas na demonstração apresentada no Capítulo 6. Pode ser encontrado também a resposta da consulta do histórico de notificações conforme citado na Seção 6.4

Listing A.1 – Dados contidos na requisição para criar um novo *template* utilizada na demonstração do Capítulo 6

```
1  {
2    "title": "Template de boas-vindas a Acme",
3    "layouts": {
4      "email": {
5        "channel": "email",
6        "type": "element",
7        "elements": [
8          {
9            "type": "meta",
10           "title": "Bem-vindo a Acme!"
11         },
12         {
13           "type": "image",
14           "src": "https://i.imgur.com/anpeUv6.jpeg"
15         },
16         {
17           "type": "text",
18           "content": "Olá, {{recipient.profile.name}}!",
19           "textStyle": "h1"
20         },
21         {
22           "textStyle": "text",
23           "bold": true,
24           "type": "text",
25           "content": "Seja bem-vindo a Acme! Estamos felizes em
                te ter no time!"
26         },
```

```
27        {
28          "textStyle": "text",
29          "type": "text",
30          "italic": true,
31          "content": "Assita o video abaixo para conhecer nossos
               valores."
32        },
33        {
34          "type": "action",
35          "content": "Assistir",
36          "href": "https://youtu.be/dQw4w9WgXcQ"
37        }
38      ]
39    },
40    "push": {
41      "channel": "push",
42      "type": "element",
43      "elements": [
44        {
45          "type": "meta",
46          "title": "Bem-vindo a Acme!"
47        },
48        {
49          "type": "text",
50          "content": "Olá, {{recipient.profile.name}}!",
51          "textStyle": "h1"
52        },
53        {
54          "type": "text",
55          "content": "Seja bem-vindo a Acme! Estamos felizes em
               te ter no time!",
56          "textStyle": "text"
57        },
58        {
```

```
59              "type": "text",
60              "content": "Acesse o app para conhecer nossos valores."
                  ,
61              "textStyle": "text"
62            }
63          ]
64        }
65      }
66  }
```

Listing A.2 – Dados contidos na requisição para criar um novo *workflow* utilizada na demonstração do Capítulo 6

```
1  {
2          "name": "Onboarding de novos membros"
3  }
```

Listing A.3 – Dados contidos na requisição para definir as etapas de um *workflow* conforme utilizada na demonstração do Capítulo 6

```
1  {
2    "name": "Onboarding de novos membros",
3    "steps": [
4      {
5        "metadata": {
6          "name": "Enviar email de boas-vindas"
7        },
8        "type": "send",
9        "channel": "email",
10       "message": {
11         "type": "template",
12         "templateId": "2NfAW0NANYwCggOErbcLuyxGYws",
13         "flow": {
14           "type": "simple",
15           "channel": "email"
16         }
17       }
```

```
18      },
19      {
20        "metadata": {
21          "name": "Enviar push de boas-vindas"
22        },
23        "type": "send",
24        "channel": "push",
25        "message": {
26          "type": "template",
27          "templateId": "2NfAW0NANYwCggOErbcLuyxGYws",
28          "flow": {
29            "type": "simple",
30            "channel": "email"
31          }
32        }
33      },
34      {
35        "metadata": {
36          "name": "Aguardar 5 minutos"
37        },
38        "type": "delay",
39        "delay": {
40          "type": "duration",
41          "unit": "minutes",
42          "value": 5
43        }
44      },
45      {
46        "metadata": {
47          "name": "Enviar boas-vindas marketing"
48        },
49        "type": "send",
50        "channel": "email",
51        "message": {
```

```
        "type": "template",
        "templateId": "2NfC7X6qQ9FCyRFkNXZ0C7Tr5zF",
        "flow": {
          "type": "simple",
          "channel": "email"
        }
      },
      "conditions": [
        {
          "op": "and",
          "conditions": [
            {
              "property": "recipient.profile.role",
              "op": "equal",
              "value": "marketing"
            }
          ]
        }
      ]
    },
    {
      "metadata": {
        "name": "Enviar boas-vindas developer (email)"
      },
      "type": "send",
      "channel": "email",
      "message": {
        "type": "template",
        "templateId": "2NfC0rndebA7ftKZEzMp4T4uQyc",
        "flow": {
          "type": "simple",
          "channel": "email"
        }
      },
```

```
86        "conditions": [
87          {
88            "op": "and",
89            "conditions": [
90              {
91                "property": "recipient.profile.role",
92                "op": "equal",
93                "value": "developer"
94              }
95            ]
96          }
97        ]
98      },
99      {
100       "metadata": {
101         "name": "Enviar boas-vindas developer (SMS)"
102       },
103       "type": "send",
104       "channel": "sms",
105       "message": {
106         "type": "template",
107         "templateId": "2NfC0rndebA7ftKZEzMp4T4uQyc",
108         "flow": {
109           "type": "simple",
110           "channel": "sms"
111         }
112       },
113       "conditions": [
114         {
115           "op": "and",
116           "conditions": [
117             {
118               "property": "recipient.profile.role",
119               "op": "equal",
```

```
              "value": "developer"
            }
          ]
        }
      ]
    }
  ]
}
```

Listing A.4 – Resposta da requisição de consulta do histórico de notificações enviadas através da execução do *workflow* construído na demonstração do Capítulo 6

```
{
  "items": [
    {
      "id": "2NfMhX9eseGeHscmMxQ3cSfC9rf",
      "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
      "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
      "recipientId": "alencar",
      "status": "undelivered",
      "channel": "email",
      "sentAt": "2023-03-29T00:46:43.000Z",
      "events": [
        {
          "id": "2NfMhgpMogMOrsLDqxEoY1VIiiG",
          "type": "undelivered",
          "ocurredAt": "2023-03-29T00:46:44.000Z",
          "data": {
            "error": {
              "type": "RecipientOptedOut",
              "message": "Recipient ’alencar’ opted out to topic
                 linked to template 2NfC0rndebA7ftKZEzMp4T4uQyc"
            }
          },
          "notificationId": "2NfMhX9eseGeHscmMxQ3cSfC9rf"
```

```
23        }
24      ]
25    },
26    {
27      "id": "2NfMhUAPsTf8uakGQZWpF0M1L17",
28      "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
29      "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
30      "recipientId": "alencar",
31      "status": "delivered",
32      "channel": "sms",
33      "sentAt": "2023-03-29T00:46:43.000Z",
34      "events": [
35        {
36          "id": "2NfMhiTyBOAj3FqSXrv2fzLruTX",
37          "type": "queued",
38          "ocurredAt": "2023-03-29T00:46:44.000Z",
39          "notificationId": "2NfMhUAPsTf8uakGQZWpF0M1L17"
40        },
41        {
42          "id": "2NfMhzr1kVFQtdCxVUUcM3htyYx",
43          "type": "sent",
44          "ocurredAt": "2023-03-29T00:46:47.000Z",
45          "notificationId": "2NfMhUAPsTf8uakGQZWpF0M1L17"
46        },
47        {
48          "id": "2NfMikyGhqNNIlvSVkkdHZjJvOd",
49          "type": "delivered",
50          "ocurredAt": "2023-03-29T00:46:53.000Z",
51          "notificationId": "2NfMhUAPsTf8uakGQZWpF0M1L17"
52        }
53      ]
54    },
55    {
56      "id": "2NfM61TIut63iVfGcNujSwZVDjX",
```

```
57      "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
58      "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
59      "recipientId": "alencar",
60      "status": "sent",
61      "channel": "push",
62      "sentAt": "2023-03-29T00:41:44.000Z",
63      "events": [
64        {
65          "id": "2NfM66qJDupUcrDrKnuR1FnZbYp",
66          "type": "queued",
67          "ocurredAt": "2023-03-29T00:41:45.000Z",
68          "notificationId": "2NfM61TIut63iVfGcNujSwZVDjX"
69        },
70        {
71          "id": "2NfM6TZxbMFK4YrUqvN1kyel3tf",
72          "type": "sent",
73          "ocurredAt": "2023-03-29T00:41:48.000Z",
74          "notificationId": "2NfM61TIut63iVfGcNujSwZVDjX"
75        }
76      ]
77    },
78    {
79      "id": "2NfM5zxcN16BL3ReN2Kvi4js2nM",
80      "workloadId": "2NfM5omJrYt9bk7NnmFjYWPJVOK",
81      "workflowId": "2NfCiGcSQxs584w5pRWtsFQLbie",
82      "recipientId": "alencar",
83      "status": "seen",
84      "channel": "email",
85      "sentAt": "2023-03-29T00:41:44.000Z",
86      "events": [
87        {
88          "id": "2NfM63HSkWPpSIBHLzQfb7MOm6t",
89          "type": "queued",
90          "ocurredAt": "2023-03-29T00:41:45.000Z",
```

```
91            "notificationId": "2NfM5zxcN16BL3ReN2Kvi4js2nM"
92          },
93          {
94            "id": "2NfM6CwyFuUJNy2lXNmFeEBLETu",
95            "type": "sent",
96            "ocurredAt": "2023-03-29T00:41:46.000Z",
97            "notificationId": "2NfM5zxcN16BL3ReN2Kvi4js2nM"
98          },
99          {
100           "id": "2NfM87tGswrfY1kbYMADQvhmKJg",
101           "type": "delivered",
102           "ocurredAt": "2023-03-29T00:42:01.000Z",
103           "notificationId": "2NfM5zxcN16BL3ReN2Kvi4js2nM"
104         },
105         {
106           "id": "2NfMN5DQhrNHH8pneANSUqWExJ3",
107           "type": "seen",
108           "ocurredAt": "2023-03-29T00:44:00.000Z",
109           "notificationId": "2NfM5zxcN16BL3ReN2Kvi4js2nM"
110         }
111       ]
112     }
113   ]
114 }
```

ANEXO B — DOCUMENTAÇÕES DA API

Através dos *links* a seguir, as documentações da API REST desenvolvida podem ser acessadas. Estas documentações possuem a descrição de cada uma das rotas, dos seus parâmetros e dados necessários, bem como exemplos de requisições. As coleções do Postman associadas a cada documentação podem ser acessadas através do botão "Run in Postman" no canto superior direito.

- Documentação das rotas de gerenciamento voltadas para uma aplicação *front-end*: <https://documenter.getpostman.com/view/25973790/2s93RMTumW>;
- Documentação das rotas de envio de notificações voltadas para outros *softwares*: <https://documenter.getpostman.com/view/25973790/2s93RMTumZ>.

ANEXO C — FORMULÁRIO DA PESQUISA

Formulário de Participação

Este formulário foi construído com a intenção de coletar a avaliação de possíveis usuários para os testes e experimentos realizados com a solução de um sistema de notificações, o qual foi desenvolvido por **Alencar da Costa** para o Trabalho de Graduação II intitulado "**Notifications as a Service: API para envio de notificações através de múltiplos canais e provedores**" do curso de Engenharia de Computação da Universidade Federal do Rio Grande do Sul (UFRGS).

A primeira seção deste questionário deve ser respondida antes da participação nos testes e experimentos. Enquanto a segunda seção deve ser realizadas após o término das tarefas propostas. A seguir uma breve descrição do conteúdo de cada seção:

- **Seção 1 - Pesquisa Demográfica:** serão coletados dados como idade, escolaridade e experiências prévias com o contexto do trabalho.
- **Seção 2 - Avalição dos Experimentos e da Solução:** serão coletados dados sobre as percepções dos usuários em relação às tarefas desenvolvidas durante os experimentos, bem como a análise sobre usabilidade da API do sistema de notificações.

Os testes e experimentos que serão realizados com o apoio deste questionário visam avaliar a usabilidade de um sistema de notificações desenvolvido com o objetivo de agregar canais e provedores e, assim, permitir o envio de notificações. Um conjunto de tarefas serão dadas para que sejam realizadas utilizando a API do sistema. Para auxiliar nas tarefas, documentações da API serão fornecidas aos participantes. Estas tarefas serão realizadas de maneira síncrona.

A seguir são apresentados alguns termos úteis utilizados durante o formulário:

- **Canal:** o meio ou canal de distribuição de notificações, por exemplo, SMS, e-mail e *push notifications.*
- **Provedor:** serviço oferecido por uma empresa para realizar o envio de notificações através de um canal específico, por exemplo, SendGrid para e-mail e Twilio para SMS.

Todo e qualquer dado obtido será utilizado única e exclusivamente para o fim a que essa pesquisa se propõe.

Duração aproximada do formulário: 5 minutos.

Nesta seção da pesquisa serão coletados dados como idade, escolaridade e experiências prévias com o contexto do trabalho.

1. Qual o seu gênero? *

- Feminino
- Masculino
- Prefiro não dizer
- Outro:

2. Qual sua faixa etária? *

- até 19 anos
- de 20 a 24 anos
- de 25 a 29 anos
- de 30 a 34 anos
- de 35 a 39 anos
- de 40 a 49 anos
- de 50 a 59 anos
- 60 anos ou mais

3. Qual o seu nível de escolaridade? *

- Ensino Médio Incompleto
- Ensino Médio Completo
- Ensino Superior Incompleto
- Ensino Superior Completo
- Mestrado ou doutorado incompleto
- Mestrado ou doutorado completo

4. Qual a sua principal atuação no momento? *

- Desenvolvedor ou desenvolvedora Front-end
- Desenvolvedor ou desenvolvedora Back-end
- Desenvolvedor ou desenvolvedora Mobile
- Educador ou educadora da área de computação
- Estudante da área de computação
- Líder Técnico
- Outro:

5. Já desenvolveu alguma integração com uma API REST? Por exemplo, realizar requisições para uma API REST utilizando alguma linguagem de programação. *

- Sim
- Não

6. Caso já tenha desenvolvido, o quão experimente você é no desenvolvimento de integrações com APIs REST?

Totalmente inexperiente

1 ○

2 ○

3 ○

4 ○

5 ○

Totalmente experiente

7. Já utilizou ou sentiu a necessidade de utilizar um provedor para envio de algum tipo de notificação para destinatários? Por exemplo, SendGrid, Twilio, Mailgun, OneSignal, Firebase FCM, AWS SES, etc.? *

○ Sim

○ Não

8. Caso você precisasse implementar o envio de notificações através de múltiplos canais, o que você escolheria? *

Sistema de notificações gerenciado: uma solução totalmente gerenciada com uma API única para todos canais e provedores de envio.

Solução própria: construir uma solução própria que realize a integração com os diferentes canais e provedores através de múltiplas APIs diferentes.

○ Dar preferência para um sistema de notificações gerenciado

○ Construir sua solução própria

Avalição dos Experimentos e da Solução

Nesta seção serão coletados dados sobre as percepções dos usuários em relação às tarefas desenvolvidas durante os experimentos, bem como a análise sobre usabilidade da API do sistema de notificações.

9. As funcionalidades do sistema de notificações permitem que eu foque no desenvolvimento das regras de negócio do meu *software* sem precisar alocar muito tempo para implementar o envio de notificações e os recursos relacionados. *

Discordo totalmente

1 ○

2 ○

3 ○

4 ○

5 ○

Concordo totalmente

10. A composição de *templates* através dos blocos de conteúdo chamados de *elements* permite que usuários sem conhecimentos técnicos construam seus próprios *templates* através de uma UI sem auxílio de desenvolvedores. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

11. Através das capacidades da API para gerenciar preferências de destinatários, eu posso, facilmente, dar poder aos meus usuários para decidirem quais conteúdos desejam receber. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

12. Para a tarefa de envio de notificações, eu escolheria desenvolver uma integração com a API do sistema de notificações ao invés de construir uma solução própria com múltiplos canais e provedores. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

13. A API de envio de notificações é desnecessariamente complexa. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

14. É fácil enviar notificações através da API. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

15. Desenvolvedores conseguirão implementar integrações com a API de envio notificações (Notifications API) **rapidamente**. A integração neste caso não diz respeito a API voltada para o *front-end*. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

16. A API tem potencial para agilizar a construção de fluxos de envio de notificações personalizados sem a necessidade de desenvolvimento extra. *

Discordo totalmente

1 ○

2 ○

3 ○

4 ○

5 ○

Concordo totalmente

17. A API do sistema de notificações é consistente no formato dos dados das requisições e respostas. *

Discordo totalmente

1 ○

2 ○

3 ○

4 ○

5 ○

Concordo totalmente

18. O padrão utilizado para o caminho (*path*) das rotas da API do sistema de notificações é consistente e intuitivo. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

19. As respostas da API em casos de erro são úteis e informativas. Por exemplo, em casos de erro de validação. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

20. A API do sistema de notificações segue as especificações e melhores práticas de uma API REST. Por exemplo, semântica das operações baseada em métodos HTTP, recursos nomeados no plural e sem verbos, paginação, sintaxe consistente, etc. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

21. A documentação da API é completa e clara. Por exemplo, contém descrições, detalhes do formato dos dados e exemplos. *

Discordo totalmente

1 ◯

2 ◯

3 ◯

4 ◯

5 ◯

Concordo totalmente

22. *Opcional.* Espaço livre para dar sugestões ou comentar algum ponto negativo ou positivo que tenha notado.

Sua resposta