UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

PEDRO WEBER MENDONÇA

Skate King: Uma aplicação geolocalizada para mapeamento de pontos de interesse

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

Orientador: Profa. Dra. Renata Galante

Porto Alegre
2022

“Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius — and a lot of courage — to move in the opposite direction.”

— E. F. SCHUMACHER

AGRADECIMENTOS

Gostaria de agradecer primeiramente aos meus pais, Máris e Paulo, por todas dificuldades que passaram para que eu tivesse condições de estudar e concluir minha graduação. Agradeço à eles também, assim como à minha irmã, Mariana, por todo apoio e suporte durante todos esses anos e que foram essenciais para que eu superasse todos os obstáculos.

Quero agradecer também à minha namorada, Clara, que foi minha fortaleza durante os anos da minha formação e durante a escrita desse trabalho. Ela que me acompanha há quase 8 anos, nos momentos bons e ruins, e é meu orgulho e inspiração.

Agradeço também aos meus colegas, em especial Lorenzo, Gustavo e Demétrio, que tornaram todas as dificuldades do curso mais leves e divertidas.

E por fim, um agradecimento especial à professora Renata Galante, que aceitou me orientar nesse trabalho e foi muito solícita durante todo o processo, respondendo até as dúvidas enviadas nos sábados à noite.

Muito obrigado.

RESUMO

Com a modernização dos smartphones e das tecnologias utilizadas por dispositivos móveis, surgem cada vez mais aplicações que utilizam dados de geolocalização para melhor atender seus usuários. Da mesma forma, o skate recentemente se tornou um esporte olímpico e sua popularidade vem crescendo cada vez mais. Com isso, surge uma oportunidade para o desenvolvimento de uma solução para fomentar a comunidade e incentivar a prática do esporte. Esse trabalho visa documentar o desenvolvimento da aplicação Skate King, uma plataforma onde skatistas, utilizando dados de geolocalização, podem cadastrar locais propícios para a prática do skate. É possível navegar por um mapa e visualizar os locais cadastrados nas proximidades, assim como avaliar, fazer check-in, e adicionar novos locais. Essa plataforma foi implementada utilizando melhores práticas de desenvolvimento de software, tanto para o backend quanto para o frontend. Um dos principais pilares para guiar o desenvolvimento é a escalabilidade, já que o objetivo é que essa solução seja utilizada por milhares de skatistas. Para isso, são aproveitados os benefícios da computação em nuvem e seguidos os princípios específicos desse contexto. Dessa forma, o resultado do desenvolvimento é uma aplicação que proporciona uma experiência fluida para os usuários e tem uma boa performance, mesmo com um grande volume de tráfego. Os princípios e padrões adotados no desenvolvimento desse projeto poderão ser seguidos em outras aplicações que visem escalabilidade e agilidade.

Palavras-chave: Skate, Nuvem, Geolocalização, App, iOS, Android.

Skate King: A geolocation application for point of interest mapping

ABSTRACT

With the modernization of smartphones and the technologies used by mobile devices, more and more applications use geolocation data to better serve their users. Likewise, skateboarding has recently become an Olympic sport and its popularity is growing each day. With this, an opportunity arises for the development of a solution to foster the community and encourage the practice of the sport. This work aims to document the development of the Skate King application, a platform where skaters, using geolocation data, can register suitable places for skateboarding. They can browse a map and view nearby registered locations, as well as rate, check-in, and add new locations. This platform was implemented using software development best practices, both for the backend and for the frontend. One of the main pillars to guide development was scalability, as the goal is for this solution to be used by thousands of skaters. For this, the benefits of cloud computing were taken advantage of and the specific principles of this context were followed. Thus, the result of development was an application that provides a fluid experience for users and performs well, even with a large volume of traffic. The principles and standards adopted in the development of this project can be followed in other applications aimed at scalability and agility.

Keywords: Skate, Cloud, Geolocation, App, iOS, Android.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
APP	Application
HTTP	Hypertext Transfer Protocol
SDK	Software Development Kit
SSO	Single Sign On
DDD	Domain Driven Design
AWS	Amazon Web Services
CRUD	Create, Retrieve, Update, Delete
JSON	Javascript Object Notation
UFRGS	Universidade Federal do Rio Grande do Sul

SUMÁRIO

1 INTRODUÇÃO

Com a recente adição do skate ao grupo de esportes olímpicos, a popularidade do esporte vem crescendo cada vez mais (IOC, 2020). O skate foi criado na década de 50 nos Estados Unidos por surfistas que queriam continuar suas atividades mesmo quando as condições marítimas não eram ideais (MARTIN, 2005). Dessa forma, cidades se tornaram grandes *playgrounds*, já que mesmo o mais simples dos locais, como um corrimão ou escada, pode ser tornar um obstáculo atrativo para skatistas. Esses locais, muitas vezes chamados de *picos*, costumam ser mapeados por skatistas locais que conhecem a cidade.

Com o advento dos *smartphones* e os avanços das tecnologias disponíveis para dispositivos móveis, o uso de informações de geolocalização em aplicativos móveis vem se tornando cada vez mais comum (ESTES, 2016). Com o uso dessas tecnologias, podem ser criadas aplicações que resolvem problemas práticos do dia-a-dia de muitas pessoas, como locomoção, rotas, descoberta de restaurantes ou estabelecimentos comerciais, entre outros. Utilizando aplicações que trabalham com geolocalização um usuário pode, por exemplo, encontrar o hospital mais próximo em situações de emergência. Juntamente com esses benefícios surge, é claro, uma preocupação com a privacidade e proteção dos dados de usuários.

A prática do skate pode se dar nos mais variados locais dentro da cidade. Desde pistas destinadas especificamente para a prática do esporte até corrimões ou escadas no contexto urbano (MACHADO, 2011). Dessa forma, a catalogação e descoberta desses locais, chamados normalmente de *picos*, torna-se uma tarefa muito importante para *skatistas*. Com um mapeamento completo e detalhado dos *picos* de uma cidade, praticantes do skate podem descobrir novos *picos* na sua cidade. Da mesma forma, skatistas visitantes em novas cidades podem encontrar e conhecer os *picos* locais. Além disso, a possibilidade de que os skatistas cadastrem e avaliem os *picos* de uma cidade também é de grande valor.

Dentro desse contexto, algumas aplicações oferecem soluções que chegam perto da resolução desses problemas. O *Google Maps* oferece uma solução completa de mapeamento de estabelecimentos, além da funcionalidade de navegação. O jogo *Pokémon GO* oferece uma experiência baseada em geolocalização com descoberta de pontos de interesse. Os aplicativos Hubba Skate Spots, SK8 Spots, Loke e Smap oferecem soluções de cadastro e descoberta de pontos de interesse focados no contexto do skate. No entanto, nenhum desses sistemas fornece uma solução completa para o problema, seja por não

terem um foco na comunidade do skate ou por não oferecerem algumas funcionalidades que são importantes para os skatistas.

Com isso, o objetivo desse trabalho é a implementação de um sistema completo para mapeamento e descoberta de *picos* para skatistas. O sistema desenvolvido conta com um aplicativo para dispositivos móveis pelo qual é possível cadastrar *picos*, assim como descobrir novos *picos* cadastrados por outros usuários. Também é possível, no aplicativo, realizar um *check-in* em um pico e o avaliar com uma nota de 1 a 5. Além disso, há um módulo gerencial destinado ao acompanhamento e sanitização dos picos *cadastrados*. A aplicação *backend* e o módulo gerencial foram desenvolvidos exclusivamente pelo autor, e a aplicação *mobile* (aplicativo) teve contribuições de um segundo desenvolvedor em partes da implementação. Foi realizado um experimento com usuários que identificou que o aplicativo implementado é simples e intuitivo. Com o suporte dos resultados do experimento também foi possível identificar melhorias importantes para as próximas versões do aplicativo. Esse trabalho documenta todo o desenvolvimento e implantação do sistema, definindo boas práticas para a criação e manutenção de sistemas de software. O processo descrito no trabalho pode servir como base para desenvolvedores que queiram implementar sistemas escaláveis de forma rápida e eficiente.

O restante do trabalho está organizado em 6 diferentes capítulos. O Capítulo 2 define conceitos e ferramentas relevantes para o sistema e para o entendimento do trabalho. O Capítulo 3 apresenta os principais trabalhos relacionados. O Capítulo 4 documenta o desenvolvimento de todas as aplicações implementadas para compor o sistema completo, assim como o racional por trás de cada decisão de projeto. O Capítulo 5 mostra as principais funcionalidades do sistema e o seu funcionamento. O Capítulo 6 apresenta o experimento realizado e os seus resultados. No Capítulo 7, por fim, é feita uma síntese do trabalho e são definidos os próximos passos para a evolução do sistema.

2 FUNDAMENTAÇÃO TEÓRICA

Neste Capítulo, constam os conceitos, tecnologias e métodos importantes para o trabalho e para a compreensão dos próximos Capítulos. São apresentados tópicos sobre metodologias ágeis e padrões de desenvolvimento de *software*, assim como linguagens de programação e tecnologias específicas.

2.1 Histórias de Usuário

Em desenvolvimento de *software* e métodos ágeis, uma história de usuário (PATTON et al., 2014) é uma descrição informal de uma ou mais funcionalidades do sistema escrita do ponto de vista do usuário final. Essa descrição tem o intuito de explicitar a forma como uma funcionalidade traz valor para os usuários. Por mais que o conceito de histórias de usuário possa parecer semelhante ao conceito de requisitos de *software*, o objetivo não é o mesmo. Histórias de usuário colocam, como pregado por diversas metologias ágeis, o usuário no centro da discussão. A descrição das histórias deve utilizar uma linguagem não técnica e deixar claro para o time o motivo de estarem construindo o sistema.

A definição de funcionalidades usando esse método visa deixar claro a resposta para as perguntas "quem?", "o que?" e "por que?" na descrição de um sistema. Para isso, muitas vezes é utilizado um *template* para a escrita das histórias:

Como < tipo de usuário >, quero < objetivo > para que < motivação >.

Dessa forma, com descrições claras das funcionalidades do sistema, fica explícito quem são os usuários finais, quais são seus objetivos com o sistema e suas motivações para utilizá-lo.

2.2 Scrum

Scrum (SCHWABER; SUTHERLAND, 2020) é um *framework* de desenvolvimento ágil que ajuda times a produzirem valor para um produto ou organização. No *Scrum*, o trabalho a ser realizado é dividido e ordenado em um *Backlog de Produto*. O time de desenvolvimento transforma uma porção do *backlog* em um incremento de valor durante uma *Sprint*. Uma *Sprint* é um ciclo de desenvolvimento de tempo fixo. Ao fim de

cada *Sprint*, o time finaliza uma entrega de valor. O *framework* define algumas práticas que podem auxiliar o time a trabalhar de forma ágil e adaptativa. No entanto, a definição do *Scrum* é propositalmente incompleta. Isso porque há o entendimento de que o propósito do *framework* é guiar as interações e relacionamento entre as pessoas que o utilizam, ao invés de criar regras fixas e detalhadas.

No início de uma *Sprint*, é realizada uma reunião de planejamento onde são definidas as prioridades do *Backlog do Produto* com ajuda do *Product Manager*. As funcionalidades priorizadas são, então, movidas para o *Backlog da Sprint*, que contém o trabalho a ser realizado durante o ciclo de desenvolvimento. Ao longo da *Sprint*, são realizadas reuniões diárias para acompanhar o progresso, identificar impedimentos ou, se necessário, repriorizar as tarefas. Essas reuniões são chamadas de *Daily Scrums*. Ao fim da *Sprint*, são realizadas duas cerimônias: *Sprint Review* e *Sprint Retrospective*. A reunião de *Sprint Review* (revisão da *sprint*) tem o objetivo de inspecionar o resultado do trabalho realizado na *sprint* e determinar adaptações futuras. A *Sprint Retrospective* (retrospectiva da *sprint*) tem o intuito de planejar formas de evoluir a produtividade e qualidade das *sprints*.

2.3 Clean Architecture

Clean Architecture (MARTIN; GRENNING; BROWN, 2018) é um padrão de desenvolvimento de software proposto por Robert Cecil Martin em seu livro em 2017. A ideia principal desse padrão é tornar os *casos de uso* o centro do sistema de software. Para isso, são definidos diversos princípios a serem considerados pelos desenvolvedores durante a criação de *softwares*. Esses princípios pregam, essencialmente, que o código destinado a lógica principal de negócio seja independente de qualquer elemento externo como bancos de dados ou bibliotecas e ferramentas. Com isso, é criada uma arquitetura em camadas, onde as camadas mais internas - e mais importantes para o sistema - contém a lógica de negócio. As camadas externas contém as implementações que mudam com mais frequência e que não tem importância para a resolução de problemas do negócio.

2.4 Domain Driven Design (DDD)

Domain Driven Design, ou DDD, é uma abordagem para criação de *softwares* que busca focar o esforço de desenvolvimento na codificação de um modelo de domínio que

contenha as regras e processos do domínio. Esse nome vem do livro escrito por Eric Evans em 2003 ((EVANS; FOWLER, 2003) que descreve essa abordagem por meio de uma série de padrões (FOWLER, 2020). Assim como o padrão Clean Architecture (Seção 2.3), um dos grandes objetivos do DDD é a minimização do esforço relacionado com resolução de problemas que não pertencem ao domínio principal do sistema. Dessa forma, o foco do desenvolvimento de um *software* é sempre a lógica de negócios que diferencia um sistema dos demais. A filosofia do DDD vai além do desenvolvimento de *software* e prega a criação de uma **linguagem ubíqua**. Isto é, uma linguagem onipresente que pode ser facilmente compreendida por pessoas de negócio ou desenvolvedores. Além disso, o *Domain Driven Design* orienta que domínios grandes sejam divididos em **contextos delimitados** menores para diminuir a complexidade total. Tanto a criação da linguagem ubíqua quanto a quebra do domínio devem ser realizadas pelos desenvolvedores em conjunto com os especialistas do domínio, que conhecem o produto e o negócio em detalhes.

2.5 12 Factor Application

12 Factor Application (WIGGINS, 2017) é uma metodologia que define algumas melhores práticas para a criação e manutenção de aplicações, principalmente no modelo **software como serviço**, ou *SaaS*. Esses princípios foram definidos por desenvolvedores da *Heroku* em 2012 em um documento publicado online. Como indicado pelo nome, esse documento define 12 fatores que têm o objetivo, entre outros, de facilitar a criação de aplicações que: automatizam a configuração inicial, são portáteis entre diferentes ambientes de execução, podem ser facilmente implantadas em plataformas de nuvem, minimizam a divergência entre desenvolvimento e produção e podem escalar sem a necessidade de mudanças significativas.

2.6 HTML

O HTML (WHATWG, 2022) é uma linguagem de marcação que define a estrutura e significado de páginas *Web*. Navegadores *Web* interpretam documentos HTML para renderizar o conteúdo das páginas. Um documento HTML é composto por texto marcado com **elementos** que definem a aparência e significado do texto. Essa marcação é feita com o uso ***tags***, que consistem em um elemento entre "<" e ">". Dessa forma, é criada

uma hierarquia de elementos que definem a estrutura de uma página. Alguns exemplos de elementos são: *title*, que define o título do documento, *body*, que define o corpo principal do documento ou *footer*, que define uma seção rodapé para o documento. Elementos de maior nível na hierarquia definem a estrutura da página, enquanto elementos mais específicos definem atributos como a formatação e aparência de uma porção de texto.

2.7 Single Sign On

Single Sign On (TERAVAINEN, 2020), ou SSO, é um mecanismo que permite que usuários possam acessar sistemas diferentes de modo transparente com apenas uma autenticação. Dessa forma, simplifica-se o controle de acessos tanto para administradores quanto para usuários. Os objetivos principais do SSO são o aprimoramento da experiência do usuário, assim como o aumento segurança e a redução dos custos com controle de acessos. Normalmente, a aplicação SSO mantém o controle da sessão principal e as aplicações alvo controlam suas sessões individualmente com base na sessão principal. Em sistemas modernos, na maioria das vezes são utilizados provedores de identidade e SSO como *Google*, *Facebook* ou *Apple*. Dessa forma, os usuários precisam administrar somente suas credenciais para esses provedores.

2.8 Javascript

O *Javascript* (ECMA International, 2022) é uma das linguagens de programação modernas mais populares. Utilizado tanto em aplicações *client-side*, como páginas web ou aplicativos móveis, quanto em sistemas *server-side*, o *Javascript* é formalmente definido pela instituição Ecma International com a especificação ECMA-262. A linguagem possui uma sintaxe alto nível que é muitas vezes chamada de orientada a protótipos, em contraste com o padrão mais conhecido de orientação a objetos. Um dos atributos mais atrativos que torna a linguagem tão popular é a tipagem dinâmica e a liberdade fornecida aos desenvolvedores. A sintaxe recebe atualizações frequentes e, desde a versão ES6, conta com abstrações e funcionalidades modernas como o uso operações assíncronas.

O Javascript vem crescendo e se tornando um dos principais componentes da internet moderna. Um dos principais fatores para isso é a dinamicidade e adaptabilidade da linguagem, que muitas vezes pode ser usada em todas as pontas de um sistema, desde

páginas *web* e aplicações *server-side*, ou até modelos de aprendizado de máquina ou análise de dados (NOOR, 2022). Além disso, a comunidade de desenvolvedores também é bastante ativa e costuma criar diversas ferramentas e bibliotecas *open source* que trazem novas funcionalidades para o ecossistema *Javascript*.

2.9 Typescript

O *Typescript* (MICROSOFT, 2012) é um *superset* da linguagem *Javascript*. Isto é, todo programa escrito corretamente em *Javascript* é também um programa válido *Typescript*. Isso porque, na verdade, o código escrito em *Typescript* é compilado para código *Javascript* que pode ser executado nos *runtimes* padrão já utilizados. O principal objetivo do *Typescript* é tornar possível a utilização do *Javascript* de forma fortemente tipada. Com isso, facilita-se a experiência dos desenvolvedores com melhores integrações com editores de textos. Erros de tipagem são reportados pelo compilador. Dessa forma, diferente da linguagem *Javascript* padrão, um código com problemas de tipagem não gera um programa que pode ser executado e não causa *bugs* em tempo de execução. Dessa forma, os desenvolvedores podem ter mais confiança no fato de que se um programa foi compilado com sucesso as chances de problemas acontecerem durante a execução são menores.

2.10 Node.js

Como mencionado anteriormente, a linguagem *Javascript* é uma das mais populares atualmente e pode ser utilizada em diversos contextos. Código *Javascript* pode ser executado em navegadores *web*, aplicações *desktop* ou em sistemas *server-side*. Cada um desses ambientes precisam de uma ferramenta que compile e execute o código. O Node.js (DAHL, 2009) é um ambiente de execução *Javascript* para sistemas que executam no servidor. Esse *runtime*, como é chamado, utiliza uma plataforma desenvolvida pelo Google chamada V8, que também é utilizada em alguns navegadores *web*. Por esse motivo, podem existir algumas diferenças na execução de código *Javascript* dependendo do ambiente em que é executado.

O *runtime Node.js* tem algumas particularidades na forma como executa o código *Javascript*. Esse ambiente utiliza um mecanismo chamado *event loop*, que enfileira inter-

namente a execução de algumas primitivas ou métodos. Com esse mecanismo, é criada uma "ilusão" de paralelismo, já que, na visão do programador, várias operações assíncronas podem ser executadas ao mesmo tempo. Por esse motivo, o Node.js é muitas vezes chamado de *pseudo-multithread*, já que na verdade há apenas uma sequência de execução.

2.11 API HTTP REST

Uma API, ou *Application Programming Interface*, é uma interface que define como os clientes de um sistema de software podem interagir com os recursos disponíveis. Esses recursos podem ser de diversos tipos como imagens, documentos HTML, metadados, objetos JSON ou XML, entre outros. Uma API HTTP usa o protocolo HTTP (*Hypertext Transfer Protocol*) como meio de comunicação entre os sistemas. API HTTP, no entanto, ainda é um conceito bastante amplo, já que esses sistemas podem ser implementados de diversas formas dependendo do contexto em que serão utilizados. Alguns exemplos de APIs HTTP são GraphQL, gRPC, SOAP ou REST. Nesse trabalho, foram utilizadas APIs REST (FIELDING, 2000).

REST significa *Representational State Transfer*, e pode ser definido basicamente como um conjunto de regras e práticas que devem ser seguidas ao compartilhar dados entre clientes e servidores. Essas regras limitam a representação de ações que podem ser realizadas no servidor a apenas *métodos* e *recursos*. Os métodos utilizados são os mesmos do protocolo HTTP, como *GET*, *POST*, *PUT* ou *DELETE*. Recursos podem ser vistos essencialmente como entidades que existem no servidor. Dessa forma, ao limitar a representação de todas as operações a métodos e recursos, o design REST obriga os desenvolvedores a modelarem as interfaces de forma simples. Assim, qualquer sistema pode ser representado por um conjunto de operações CRUD (*Create, Retrieve, Update, Delete*), por mais complexo que seja.

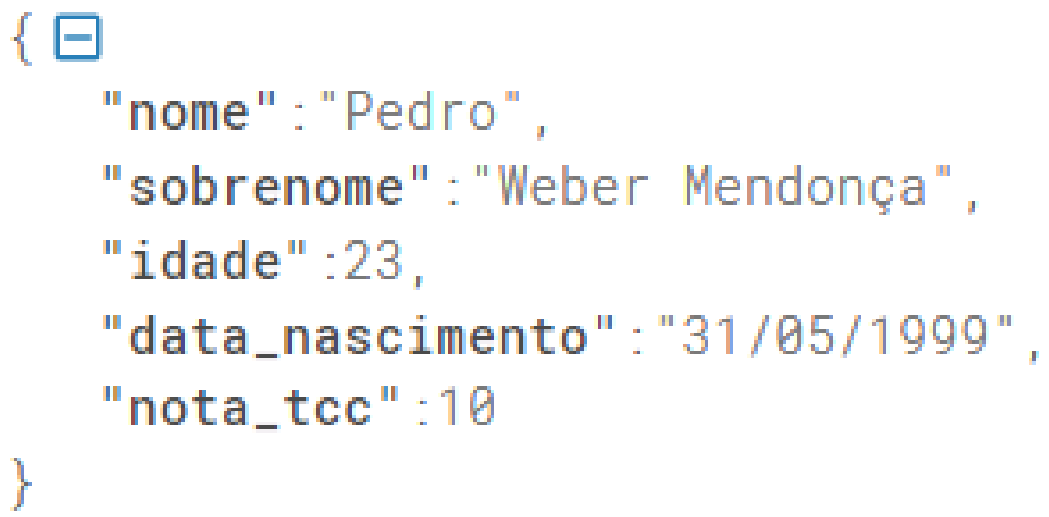
Em uma API REST que disponibiliza uma interface para compras online, por exemplo, a ação criação de um pedido para um cliente não seria representada por um método chamado *criarPedido* ou *realizarCompra*. Nesse caso, seguindo as melhores práticas de APIs REST, essa operação poderia ser representada pelo método POST (criação de um recurso) em conjunto com os recursos *usuário* e *pedido*: *POST /usuários/ID/pedidos*, onde *ID* representa o identificador único do usuário.

2.12 JSON

JSON (INTERNATIONAL, 2017) significa *Javascript Object Notation* (*Notação de Objetos Javascript*) e é um formato leve para troca de dados entre sistemas. Essa notação tem intuito de ser facilmente legível por seres humanos e, ao mesmo tempo, facilmente interpretada e gerada por computadores. Como se pode notar pelo nome, o padrão *JSON* é baseado na definição de objetos da linguagem *Javascript*.

Um objeto é definido por um conjunto de chaves e valores, iniciando com uma abertura de chave e terminando com um fechamento de chave. Cada propriedade é definida por uma chave e um valor, separados por dois pontos. A Figura 2.1 a seguir mostra um exemplo de objeto JSON com a representação dos dados de uma pessoa.

Figura 2.1 – Exemplo de objeto JSON



Fonte: Autor

2.13 Amazon Web Services

A Amazon Web Services, ou AWS, é um dos principais provedores de nuvem pública atualmente. A AWS tem uma infraestrutura global com *datacenters* espalhados entre 27 regiões em todos os continentes (AWS, 2022a). Cada região é subdividida entre **zonas de disponibilidade**, que são diferentes *datacenters* segregados fisicamente. Com essa infraestrutura, a AWS oferece mais de 200 serviços com diferentes finalidades como computação, bancos de dados, *machine learning*, entre outros. Ao utilizar os serviços da AWS, os clientes aderem ao **modelo de responsabilidade compartilhada** (AWS, 2022b)

definido pela AWS sobre a segurança das aplicações em nuvem. Esse modelo define, essencialmente, que o provedor (AWS) é responsável pela "segurança da nuvem", enquanto o cliente é responsável pela "segurança na nuvem". A segurança da nuvem consiste na proteção da infraestrutura que executa os serviços oferecidos, como *hardware*, *software*, *redes* e *instalações*. A segurança na nuvem é definida como a segurança das aplicações implantadas utilizando os serviços da nuvem, como controle de permissões ou gerenciamento dos sistemas operacionais convidados (quando existentes).

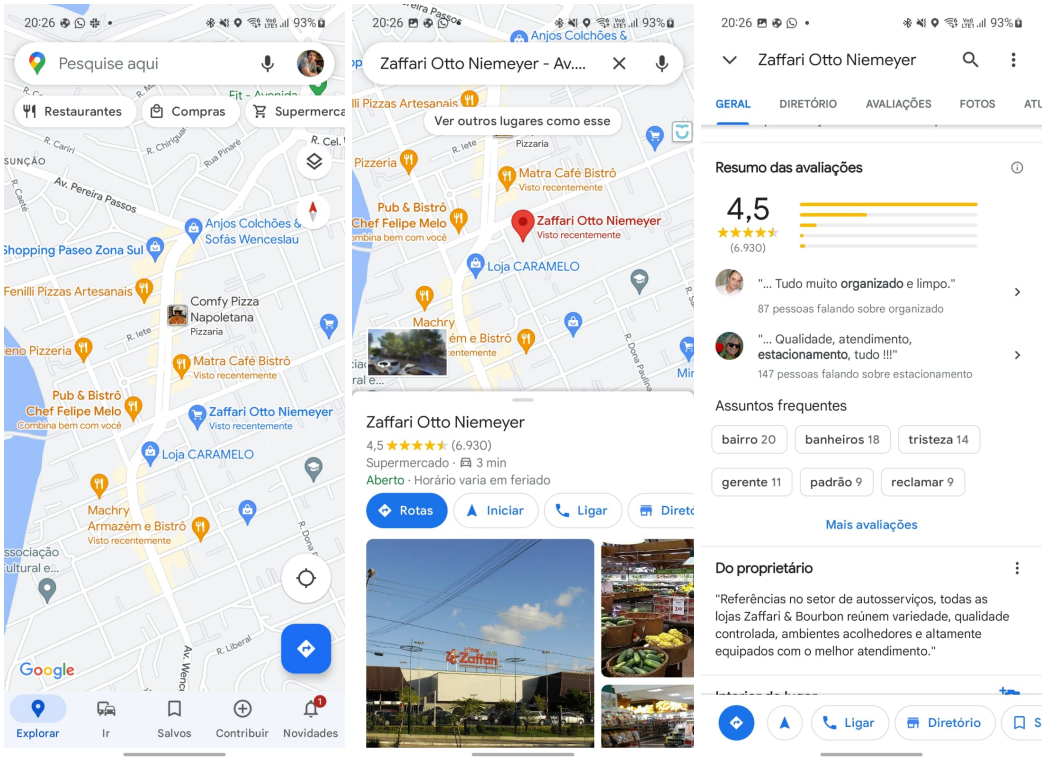
3 TRABALHOS RELACIONADOS

Neste Capítulo, são apresentados seis aplicativos móveis que compartilham características com esse projeto, seja por possuírem um foco na comunidade de skate, lidarem com dados de geolocalização ou por serem gerenciados pela própria comunidade. O primeiro é o *Google Maps*, um aplicativo de mapas bastante completo, com foco em navegação e rotas. O segundo é o aplicativo *Hubba Skate Spots*, que é destinado ao cadastro de pistas e pontos de interesse de *skate*. O terceiro é o jogo Pokemon GO, que tem como premissa o uso de realidade aumentada e geolocalização para estimular os jogadores a se deslocarem pela cidade e visitarem pontos de interesse. O quarto, quinto e sexto aplicativos são *SK8 Spots*, *Loke* e *Smap*. Todos eles tem seu foco na comunidade de skate e no mapeamento da cidade em relação a locais propícios para a prática do skate. No entanto, cada um deixa lacunas em questões específicas, seja em funcionalidades ou experiência do usuário. Ao fim do Capítulo é apresentada uma análise comparativa entre todos os trabalhos.

3.1 Google Maps

O *Google Maps* (GOOGLE, 2022) é um sistema de mapas completo com foco em navegação e rotas. Normalmente é utilizado para encontrar o caminho mais rápido até um endereço. No entanto, também existe um lado voltado para o mapeamento e descoberta de pontos de interesse da cidade. Ao navegar pelo mapa do aplicativo, é possível encontrar lojas, restaurantes e diversos outros estabelecimentos comerciais. Esses locais são representados por *pins* posicionados no mapa e, ao clicar em um *pin*, são mostradas várias informações como horário de funcionamento, telefone, fotos do local, assim como avaliações e comentários de outros usuários. Esse mapeamento, no entanto, é focado em estabelecimentos comerciais e os *pins* não podem ser cadastrados de forma simples por qualquer pessoa. A Figura 3.1 mostra o funcionamento do aplicativo.

Figura 3.1 – Telas do aplicativo *Google Maps*

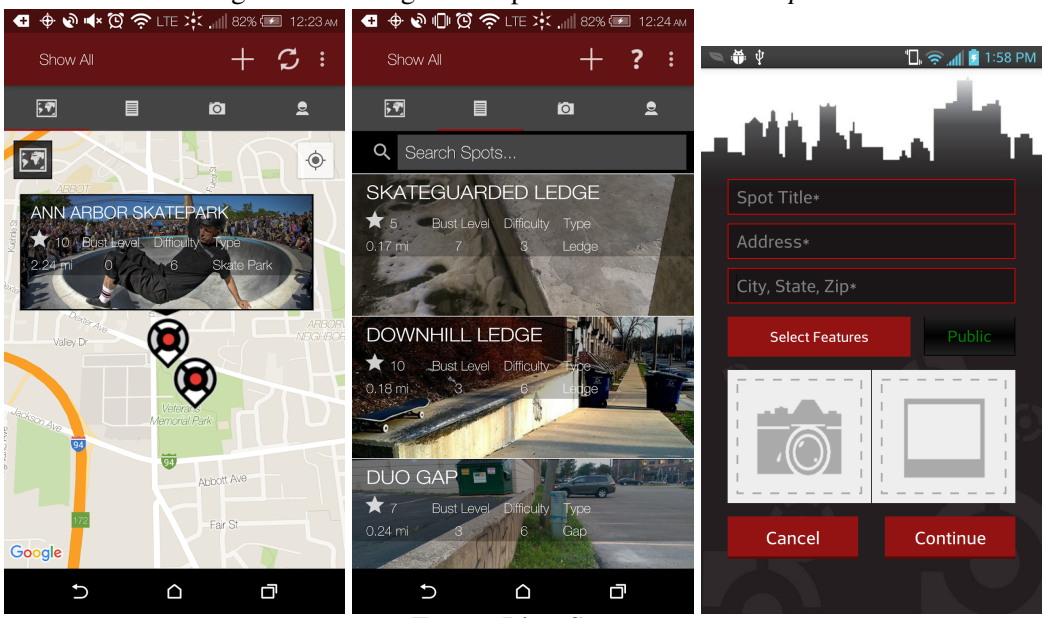


Fonte: Google Maps

3.2 Hubba Skate Spots

O *Hubba Skate Spots* (SKATE, 2017) é um aplicativo destinado ao cadastro de pontos de interesse para a comunidade de skate. O sistema permite que sejam cadastrados locais como pistas ou clubes. Além disso, é possível navegar pelo mapa e visualizar locais cadastrados por outros usuários, além de visualizar o perfil dos skatistas. No entanto, nos testes realizados, os locais demoraram mais de 1 minuto para serem carregados e o aplicativo apresentou vários erros. Por ser um aplicativo relativamente antigo, o *layout* e a experiência de uso estão defasados em relação à expectativa dos usuários, especialmente considerando as evoluções recentes nas áreas de *design* e experiência do usuário. A Figura 3.2 apresenta algumas imagens do aplicativo.

Figura 3.2 – Imagens do aplicativo *Hubba Skate Spots*



Fonte: Play Store

3.3 Pokémon GO

O *Pokémon GO* (NIANTIC, 2022) foi um dos primeiros aplicativos a popularizar o uso de tecnologias baseadas em geolocalização. Nesse aplicativo, os usuários jogam como treinadores de monstros chamados *pokémons* e devem caminhar pela rua para capturar novos *pokémons* e coletar recursos em pontos de interesse pela cidade. No entanto, esses pontos de interesse são gerenciados pela desenvolvedora do jogo e não podem ser cadastrados pelos jogadores. Na tela principal do jogo, é exibido o mapa, onde o usuário pode visualizar os pontos de interesse e andar em direção a eles. A Figura 3.3 mostra imagens do funcionamento do aplicativo.

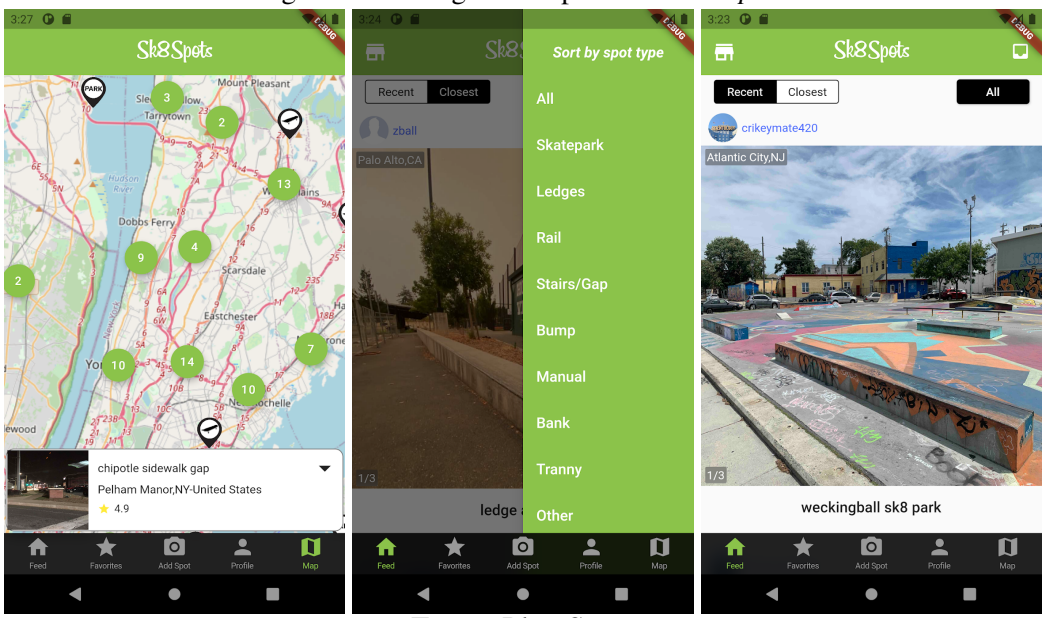
Figura 3.3 – Imagens do aplicativo *Pokémon GO*

Fonte: Play Store

3.4 SK8 Spots

SK8 Spots (ALEXANDER, 2021) é também destinado ao mapeamento e descoberta de locais propícios para a prática do skate. Em relação ao anterior, o aplicativo *SK8 Spots* é mais completo e disponibiliza funcionalidades importantes como avaliações e comentários. No entanto, a interface tem uma aparência defasada em relação à aplicativos mais modernos e a experiência é pouco intuitiva. Além disso, o aplicativo não tem a funcionalidade de *check-in* nos locais cadastrados. Na Figura 3.4 são ilustradas algumas das funcionalidades.

Figura 3.4 – Imagens do aplicativo *SK8 Spots*

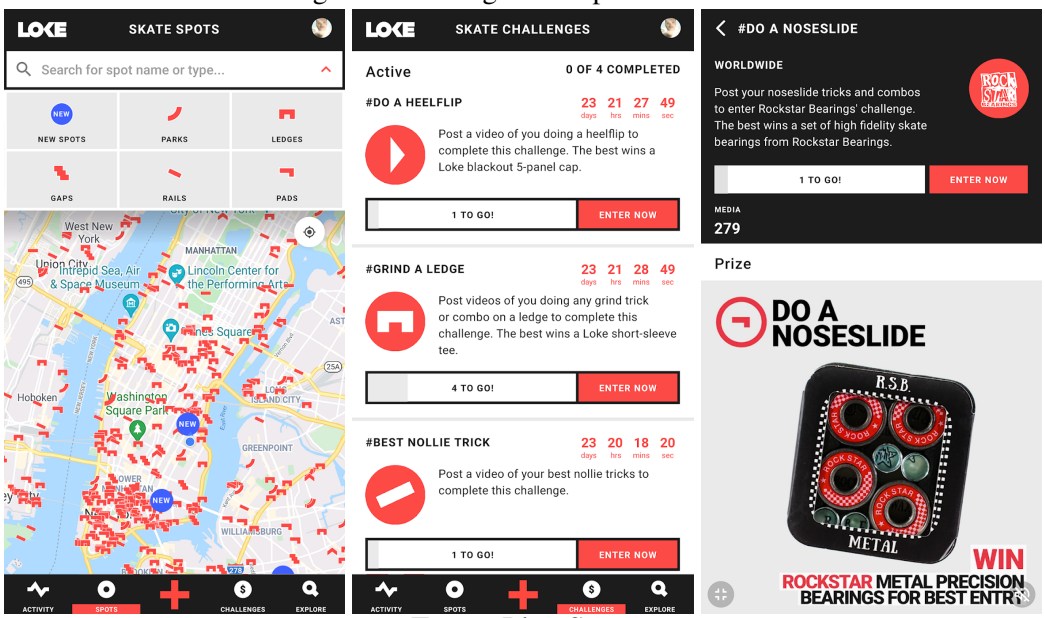


Fonte: Play Store

3.5 Loke

Loke (ATTABOY, 2022) é um aplicativo focado na comunidade de skate que tem o intuito de oferecer funcionalidades como cadastro de pontos de interesse, publicação de vídeos de skate, notificações sobre notícias do mundo do skate e desafios que podem ser completados por prêmios. O sistema também disponibiliza uma funcionalidade semelhante ao *check-in*, chamada de *session*. No entanto, por conta da grande quantidade de funcionalidades, o aplicativo é confuso e poluído, além de apresentar diversas lentidões e travamentos. A Figura 3.5 mostra algumas imagens da interface do sistema.

Figura 3.5 – Imagens do aplicativo *Loke*

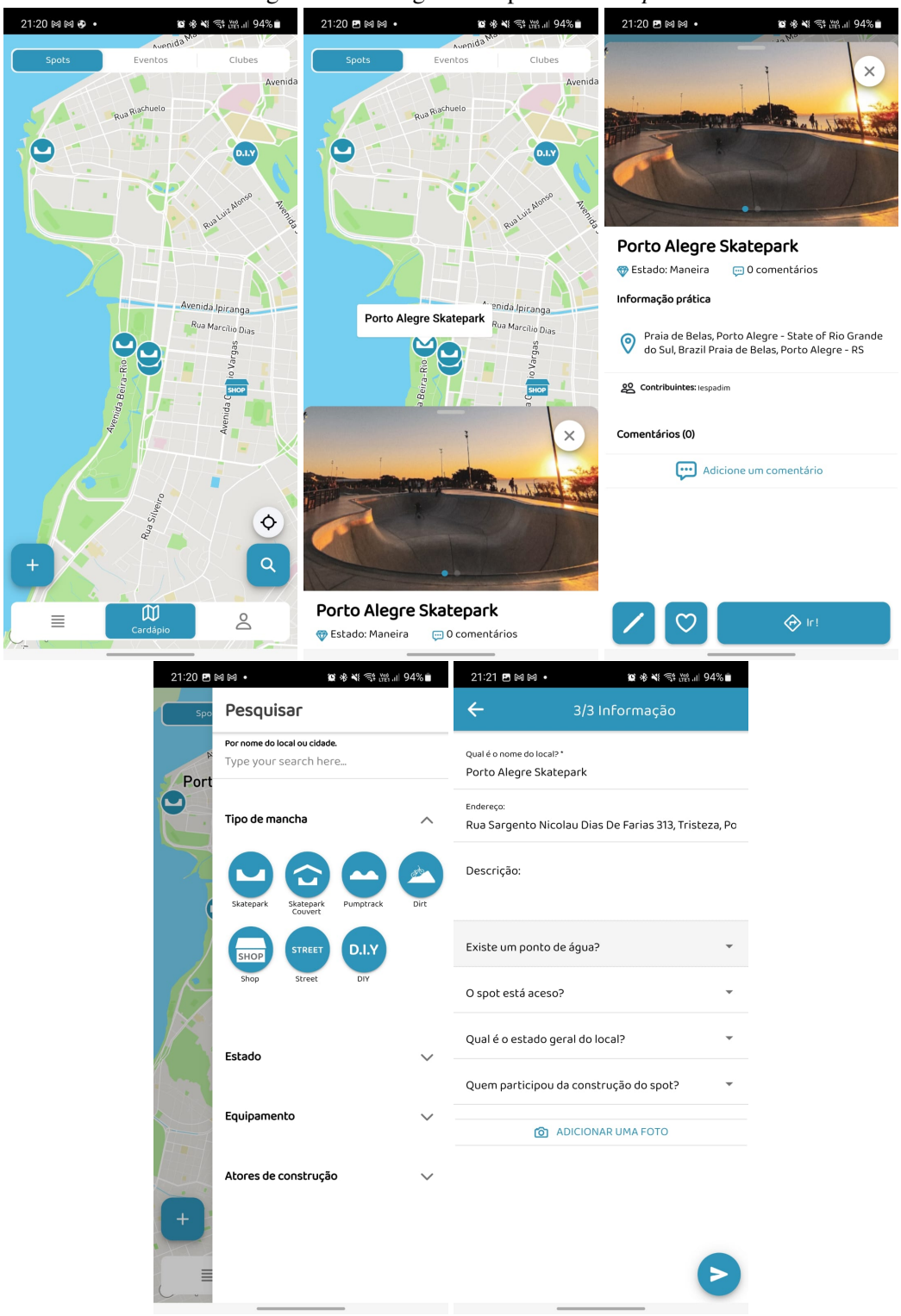


Fonte: Play Store

3.6 Smap

Dentre os aplicativos de skate apresentados, o *Smap* (ARTIUM, 2022) é o mais completo e que oferece a melhor experiência ao usuário. O foco desse sistema não é apenas o skate, mas também outras modalidades como patins, BMX ou *scooter*. São disponibilizadas funcionalidades como cadastro de pontos de interesse, busca e filtros por pontos de interesse e locais favoritos. Além disso, o visual do aplicativo é moderno e a experiência é intuitiva. No entanto, todos os cadastros de locais passam pela avaliação dos desenvolvedores e podem demorar até serem disponibilizados no mapa. Além disso, algumas funcionalidades chave como avaliações e *check-in* não estão disponíveis. A Figura 3.6 apresenta algumas telas do aplicativo.

Figura 3.6 – Imagens do aplicativo *Smap*



Fonte: Smap

3.7 Análise comparativa

Com a descração de cada um dos trabalhos, pode-se notar que existem diversas características em comum. Sejam elas o uso de mapas e geolocalização ou o foco no público que pratica o skate. No entanto, nem todos os sistemas têm o mesmo objetivo final e, dentre os que visam o mesmo objetivo, existem diferenças em funcionalidades ou qualidade da experiência do usuário. Na Tabela 3.1 são comparadas algumas características específicas entre os aplicativos.

Tabela 3.1 – Comparação de funcionalidades

Característica	**Google Maps**	**Hubba Skate Spots**	**SK8 Spots**	**Loke**	**Smap**	**Pokemon GO**
Experiência simples e fluida	Sim	Não	Não	Não	Sim	Sim
Focado na comunidade de skate	Não	Sim	Sim	Sim	Sim	Não
Baseado em geolocalização	Sim	Sim	Sim	Sim	Sim	Sim
Gerenciado pela comunidade	Parcial	Sim	Sim	Sim	Sim	Não
Avaliação de locais	Sim	Não	Sim	Não	Não	Não
***Check-in* em locais**	Não	Não	Não	Sim	Não	Sim

Fonte: Autor

Como se pode notar pela tabela, mesmo que alguns aplicativos como o *Google Maps* ou o *Smap* sejam completos e forneçam ao usuário uma boa experiência, nenhum dos sistemas preenche todos os critérios selecionados. Esse trabalho visa preencher todas as lacunas que ficam em aberto após essa análise. Além disso, apesar de alguns dos trabalhos analisados proporem uma solução semelhante, o *Skate King* busca entregar ao usuário essas funcionalidades e, ao mesmo tempo, oferecer uma experiência simples e fluida.

4 IMPLEMENTAÇÃO

Esse Capítulo descreve a metodologia utilizada para a implementação da plataforma Skate King, tanto em relação ao *front-end* quanto ao *back-end*, assim como as principais decisões tecnológicas tomadas e seus motivadores. Primeiramente, são apresentadas as principais funcionalidades propostas e a forma com que a implementação foi organizada temporalmente. Em seguida, define-se os principais pilares que guiaram o desenvolvimento do software. Entenderemos de forma geral como a arquitetura da plataforma foi organizada e veremos em detalhes a implementação de duas aplicações *front-end* e uma aplicação *back-end*. Além disso, discorre-se sobre como é possível aproximar-se dos pilares propostos com a ajuda da computação em nuvem. Por fim, são apresentadas as dificuldades que precisaram ser contornadas para lidar de forma eficiente com dados de geolocalização.

4.1 Visão Geral

As funcionalidades propostas para o sistema foram separadas em dois grandes grupos. Primeiramente, existem as funcionalidades do aplicativo, que têm seu foco no usuário final e são o centro da aplicação, trazendo o maior valor de negócio. Por outro lado, temos as funcionalidades destinadas às pessoas responsáveis por gerenciar a plataforma. Essas funcionalidades têm um caráter secundário em relação às funcionalidades do aplicativo, mas são essenciais para o pleno funcionamento do sistema.

A seguir, são descritas todas as funcionalidades utilizando o método de histórias de usuário. Cada funcionalidade foi categorizada entre ***Aplicativo*** e ***Gerencial***.

Aplicativo**:** Cadastro e Login

Como usuário, quero realizar cadastro e login de forma simples utilizando minha conta de provedores como Google ou Apple, para que possa acessar a plataforma Skate King.

Aplicativo**:** Navegação no mapa e visualização de *picos*

Como skatista, quero navegar pelo mapa do aplicativo visualizando os *picos* adicionados por outros usuários, para que possa encontrar novos locais para andar de skate.

Aplicativo**:** Criação de *pico*

Como skatista, quero cadastrar na plataforma os meus *picos* favoritos na cidade, para que outros skatistas possam encontrar e utilizar esses locais.

***Aplicativo*:** Visualização de um *pico*

Como skatista, quero visualizar informações sobre um *pico*, para que possa descobrir onde está localizado, assim como quais obstáculos existem no local.

***Aplicativo*:** Perfil de usuário

Como skatista, quero cadastrar no aplicativo minhas redes sociais e as marcas de skate que utilizo, para que possa compartilhar essas informações com outros skatistas.

***Aplicativo*:** Check-in em um *pico*

Como skatista, quero realizar um check-in quando estou em um *pico*, para que outros skatistas saibam que estou aqui e possam me acompanhar.

***Aplicativo*:** Avaliação de um *pico*

Como skatista, quero avaliar um *pico*, para que outros skatistas saibam sobre a qualidade desse local.

***Aplicativo*:** Deleção de um *pico*

Como skatista, quero deletar um *pico* que criei, para que possa remover informações erradas ou cadastradas por engano.

***Gerencial*:** Login

Como mantenedor da plataforma Skate King, quero utilizar um nome de usuário e senha para acessar o módulo gerencial, para que apenas pessoas autorizadas possam realizar ações gerenciais.

***Gerencial*:** Visualização de métricas

Como mantenedor da plataforma Skate King, quero acompanhar os números importantes sobre o uso do sistema, para que possa saber como a aplicação está evoluindo e como novas ações impactam os usuários.

***Gerencial*:** Navegação no mapa e visualização de *picos*

Como mantenedor da plataforma Skate King, quero navegar pelo mapa em uma interface web visualizando os *picos* cadastrados pelos usuários, para que possa acompanhar o crescimento da aplicação e encontrar regiões com grande uso.

***Gerencial*:** Visualização de informações de um *pico*

Como mantenedor da plataforma Skate King, quero visualizar informações sobre um pico, para que possa conhecer locais importantes e identificar cadastros mal-intencionados.

***Gerencial*:** Deleção de um *pico*

Como mantenedor da plataforma Skate King, quero deletar qualquer pico, para que possa manter o sistema livre de cadastros mal-intencionados.

***Gerencial*:** Criação de um *pico*

Como mantenedor da plataforma Skate King, quero cadastrar novos *picos* em uma interface web, para que possa pré-cadastrar *picos* a partir de dados conhecidos.

Ao todo, participaram do projeto dois desenvolvedores (o autor e um segundo desenvolvedor), um especialista em design e experiência do usuário, e três pessoas que trabalharam focadas em questões estratégicas e de produto. Com isso, tornou-se necessário utilizar alguma metodologia para organizar o trabalho entre todos os envolvidos e priorizar as entregas. Para tal, foi adotada uma metodologia baseada em metódos agéis. Na prática, o método utilizado foi bastante semelhante à metodologia ágil *Scrum*. Dessa forma, os períodos de desenvolvimento foram divididos em *time-boxes* de 3 semanas. Nesse modelo, durante cada *time-box*, o time se compromete com a entrega de funcionalidades que tenham valor de negócio. Assim como no *Scrum*, cada *time-box* de desenvolvimento foi chamado de *Sprint*.

Para que houvesse um forte comprometimento com a priorização das funcionalidades implementadas, foi adicionada uma variação à metodologia padrão do *Scrum*. As funcionalidades propostas foram separadas em pacotes, sendo que o início do desenvolvimento do próximo pacote é condicionado ao atingimento de uma meta de negócio. Dessa forma, foi possível criar uma relação entre o crescimento da aplicação do ponto de vista de funcionalidades e o crescimento do ponto de vista de negócio. A separação das funcionalidades em pacotes é descrita nas Tabelas 4.1, 4.2, 4.3, 4.4.

Tabela 4.1 – Pacote de funcionalidades 1

Meta gatilho	Funcionalidades
- N/A	**Aplicativo**: Navegação no mapa e visualização de *picos* **Aplicativo**: Criação de *pico* **Aplicativo**: Visualização de um *pico*

Fonte: Autor

Tabela 4.2 – Pacote de funcionalidades 2

Meta gatilho	Funcionalidades
- 1.000 Downloads nas lojas de aplicativos	**Aplicativo**: Cadastro e Login **Aplicativo**: Deleção de um *pico* **Gerencial**: Login **Gerencial**: Visualização de métricas

Fonte: Autor

Tabela 4.3 – Pacote de funcionalidades 3

Meta gatilho	Funcionalidades
- 5.000 Usuários cadastrados	**Aplicativo**: Perfil de usuário **Aplicativo**: Check-in em um *pico* **Gerencial**: Navegação no mapa e visualização de *picos* **Gerencial**: Visualização de informações de um *pico* **Gerencial**: Deleção de um *pico*

Fonte: Autor

Tabela 4.4 – Pacote de funcionalidades 4

Meta gatilho	Funcionalidades
- 10.000 Usuários cadastrados	**Aplicativo**: Avaliação de um *pico* **Gerencial**: Criação de um *pico*

Fonte: Autor

Considerando a duração de *sprint* escolhida e o número total de funcionalidades, o cronograma foi organizado ao longo de 5 *sprints*. Dessa forma, a duração do projeto foi estimada em 15 semanas. No entanto, como o desenvolvimento de novos pacotes de funcionalidades foi atrelado ao atingimento das metas, o tempo total efetivo entre o início e o fim do desenvolvimento das funcionalidades propostas foi maior do que 15 semanas.

A seguir, são apresentadas todas as *sprints* com seus respectivos objetivos e tarefas (Tabelas 4.5, 4.6, 4.7, 4.8, 4.9). As tarefas foram subdividas entre tarefas relacionadas ao *front-end* e ao *back-end*. As tarefas de design e negócio que são pré-requisitos para tarefas de desenvolvimento foram executadas uma *sprint* antes da tarefa em questão. As Tabelas a seguir descrevem apenas as tarefas relacionadas ao desenvolvimento das aplicações.

Tabela 4.5 – Objetivos e Tarefas da Sprint 1

Objetivos	Tarefas
- Primeiro protótipo funcional da aplicação. - Funcionalidades mínimas para validação da ideia: navegação no mapa, visualização de detalhes de *picos* e criação de *picos*.	***Front-end***: Implementação da tela do mapa. ***Front-end***: Implementação da tela de detalhes de um *pico*. ***Front-end***: Implementação da tela de cadastro de *pico*. ***Back-end***: Implementação do endpoint de listagem de *picos*. ***Back-end***: Implementação do endpoint de detalhes de um *pico*. ***Back-end***: Implementação do endpoint de criação de um *pico*.

Fonte: Autor

Tabela 4.6 – Objetivos e Tarefas da Sprint 2

Objetivos	Tarefas
- Sistema de cadastro e login para conhecer mais os usuários. - Funcionalidade de deleção de *picos*. - Versão inicial do módulo gerencial com funcionalidades mínimas: login e acompanhamento de métricas.	***Front-end***: Implementação da tela de cadastro/login federado (Google e Apple). ***Front-end***: Adição de botão para deleção do *pico* na tela de detalhes. ***Front-end***: Implementação da tela de login no módulo gerencial. ***Front-end***: Implementação da tela de métricas no módulo gerencial. ***Back-end***: Implementação do endpoint de deleção de um *pico*. ***Back-end***: Implementação do fluxo de login e cadastro. ***Back-end***: Implementação do endpoint de métricas.

Fonte: Autor

Tabela 4.7 – Objetivos e Tarefas da Sprint 3

Objetivos	Tarefas
- Funcionalidade de check-in em *picos* - Funcionalidade de perfil de usuário.	***Front-end***: Implementação da tela de perfil do usuário. ***Front-end***: Implementação da tela de edição do perfil do usuário. ***Front-end***: Adição de botão para check-in no *pico* na tela de detalhes. ***Front-end***: Adição de campo com número de check-ins na tela de detalhes. ***Back-end***: Implementação do endpoint de check-in em um *pico*. ***Back-end***: Adição de campo com número de check-ins no endpoint de detalhes de um *pico*. ***Back-end***: Implementação do endpoint de edição do perfil do usuário. ***Back-end***: Implementação do endpoint de detalhes do perfil do usuário.

Fonte: Autor

Tabela 4.8 – Objetivos e Tarefas da Sprint 4

Objetivos	Tarefas
- Evolução do módulo gerencial para incluir ações gerenciais sobre *picos*	***Front-end***: Implementação da tela de mapa no módulo gerencial. ***Front-end***: Implementação da tela de detalhes de um pico no módulo gerencial. ***Front-end***: Adição de botão para deleção um *pico* no módulo gerencial.

Fonte: Autor

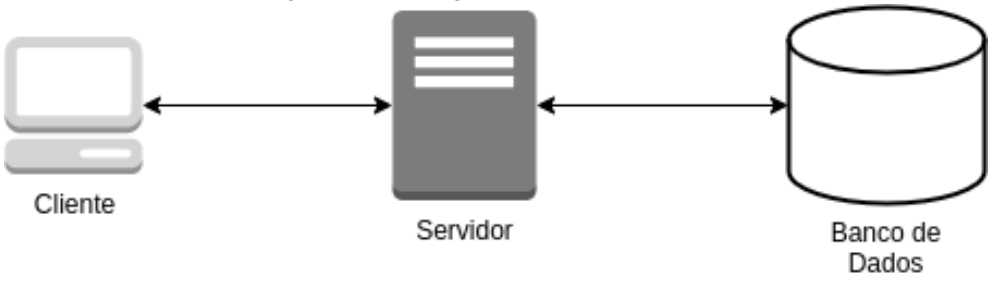
Tabela 4.9 – Objetivos e Tarefas da Sprint 5

Objetivos	Tarefas
- Funcionalidade de avaliação de *picos* - Inclusão da funcionalidade de criação de *picos* no módulo gerencial	***Front-end***: Implementação da tela de avaliações de um *pico*. ***Front-end***: Implementação da tela de criação de um *pico* no módulo gerencial ***Back-end***: Implementação de endpoint para avaliação de um *pico*. ***Back-end***: Implementação de endpoint para listagem de avaliações de um *pico*.

Fonte: Autor

Assim como grande parte das aplicações de Internet modernas, o sistema foi arquitetado utilizando como base o modelo Cliente-Servidor. Como descrito na listagem de funcionalidades, a plataforma é composta por 2 aplicações *front-end* e 1 aplicação *back-end*. O modelo Cliente-Servidor funciona muito bem nesse caso, já que as duas aplicações *front-end* podem consumir o mesmo *back-end* sem que seja necessário nenhuma adaptação. Esse tipo de arquitetura é ilustrado na Figura 4.1.

Figura 4.1 – Diagrama Cliente-Servidor.



Fonte: Autor

Dessa forma, todas as regras que são essenciais para o sistema ficam centralizadas no *back-end* (Servidor, na Figura 4.1). Mesmo que sejam implementadas novas aplicações *front-end* no futuro (Clientes, na Figura 4.1), as regras permanecem as mesmas. Além disso, como o servidor é responsável pelo controle e acesso aos dados armazenados, os usuários sempre terão acesso às mesmas informações, independentemente do dispositivo que estejam utilizando. Nas próximas Seções, entenderemos os detalhes da

implementação desse modelo e os principais pilares considerados na tomada de decisões técnicas.

4.2 Principais Pilares

Esta Seção apresenta detalhes sobre os principais pilares considerados durante o desenvolvimento. Inicia-se com uma conceituação sobre a relação entre requisitos e decisões de projeto e, em seguida, apresenta-se um aprofundamento sobre os pilares adotados e suas características.

4.2.1 Motivação

Nas últimas décadas, o mercado de tecnologia vem crescendo de forma acelerada. Com isso, surge a necessidade da definição de padrões e metodologias para que se crie software com mais eficiência. Esses métodos, que podem ser chamados de diversas formas, como *padrões de projeto*, *melhores práticas de desenvovimento*, *padrões de arquitetura ou design* entre outros, são criados e adaptados a todo momento. Frente a essa imensidão de padrões e métodos, cabe ao desenvolvedor entender os motivadores por trás de cada um deles e avaliar de que forma a adoção de um método pode beneficiar a sua aplicação. Além disso, o desenvolvedor deve ser capaz de adaptar os métodos já conhecidos para melhor atender o contexto e as particularidades do seu projeto, ao invés de seguir de forma rígida as regras definidas.

Assim, para o desenvolvimento desse trabalho, foram escolhidos alguns pilares para guiarem o desenvolvimento e as decisões a serem tomadas. Para a definição desses pilares, foram utilizados alguns padrões como inspiração. São eles: *Clean Architecture*, *12 Factor Application* e *Domain Driven Design*. Alguns princípios e objetivos desses métodos foram considerados na definição dos pilares por terem uma grande importância para essa aplicação. Esses princípios são descritos nas Tabelas 4.10, 4.11 e 4.12, a seguir.

Tabela 4.10 – Princípios importantes do padrão *Clean Architecture*

Padrão	**Princípio**
Clean Architecture	***Independência de Frameworks*:** A arquitetura do sistema não deve depender da existência de uma biblioteca ou *framework* terceiro. Essa independência possibilita que o desenvolvedor utilize essas bibliotecas como ferramentas e não como parte essencial da aplicação.
Clean Architecture	***Testabilidade*:** As regras de negócio devem ser testadas independentemente de uma interface, banco de dados, servidor web ou qualquer outro elemento externo.
Clean Architecture	***Independência do Banco de Dados*:** Deve ser possível substituir o banco de dados utilizado por outro sem que as regras de negócio sejam afetadas.

Fonte: Autor

Tabela 4.11 – Princípios importantes do padrão *12 Factor Application*

Padrão	**Princípio**
Twelve Factor Application	***Implantação em Nuvem*:** A aplicação é adequada para implantação nas modernas plataformas em nuvem, evitando a necessidade por servidores e administração do sistema.
Twelve Factor Application	***Implantação Contínua*:** A arquitetura e o design do sistema devem minimizar a divergência entre os ambientes de desenvolvimento e produção, permitindo a implantação contínua para máxima agilidade.
Twelve Factor Application	***Escalabilidade*:** A aplicação deve poder escalar sem significativas mudanças em ferramentas, arquiteturas ou práticas de desenvolvimento.

Fonte: Autor

Tabela 4.12 – Princípios importantes do padrão *Domain Driven Design*

Padrão	**Princípio**
Domain Driven Design	***Foco no Domínio*:** O foco primário da aplicação deve estar no domínio de negócio principal e na lógica de domínio.
Domain Driven Design	***Complexidade do Domínio*:** A maior parte da complexidade do sistema deve estar relacionada com a complexidade do domínio, e não com a complexidade do software (também chamada de *complexidade acidental*).

Fonte: Autor

Nas próximas Seções será detalhado cada um dos pilares escolhidos para essa implementação.

4.2.2 Escalabilidade

A escalabilidade é um atributo frequentemente reivindicado por sistemas de software modernos. Embora a noção básica seja intuitiva, não existe uma definição clara e amplamente aceita sobre o que o termo escalabilidade significa no contexto de sistemas de software. Por esse motivo, a utilização desse termo acaba se tornando um artifício de marketing ao invés de uma caracterização técnica de um sistema (HILL, 1992). Com isso, foram adotados, para esse trabalho, critérios específicos para caracterizar e medir a escalabilidade das aplicações. Esses critérios são definidos a seguir com o auxílio de uma pergunta chave para cada.

***Escalabilidade da aplicação*:** Quão fácil é adicionar uma funcionalidade ao sistema?

Esse critério foi definido pensando na longevidade da aplicação. Com o crescimento e evolução da plataforma, surgem novas ideias de funcionalidades que podem trazer valor para os usuários. Com esse critério, evita-se que o sistema chegue a um ponto em que a adição de uma nova funcionalidade traga mais custo do que valor. Por isso, é importante

que o desenvolvedor tenha em mente o impacto de uma decisão técnica na complexidade total da aplicação. O princípio de ***Foco no Domínio*** do padrão *Domain Driven Design* (Tabela 4.12) contribui fortemente para esse critério.

***Escalabilidade da infraestrutura*:** Quantas mudanças serão necessárias para que o sistema suporte dez vezes mais volume do que recebe hoje?

Esse critério foi definido também pensando na evolução da plataforma. A aplicação deve, por *design*, escalar para maiores volumes sem a necessidade de um grande esforço. Com isso, os desenvolvedores podem ter sua atenção focada na implementação de novas funcionalidades que agreguem valor para os usuários finais. Essa atenção ao que traz valor está muito alinhada com o princípio ***Complexidade do Domínio*** do padrão *Domain Driven Design* (Tabela 4.12), já que o maior esforço será direcionado a resolver problemas de negócio, e não problemas de software. Além disso, esse critério também vai claramente de encontro ao princípio de ***Escalabilidade*** do padrão *Twelve Factor Application* (Tabela 4.11).

4.2.3 Agilidade

No mundo de inovação das *startups*, o conceito de *time-to-market* é muito valorizado. O *time-to-market* consiste, de forma geral, no tempo necessário para que a solução de um problema seja implementada e disponibilizada para os usuários. Para que esse tempo seja minimizado, uma nova ideia pode ser validada de forma rápida e sem grande esforço com uma versão simplificada da solução. Dessa forma, iniciativas com grande custo não são implementadas com base em uma suposição e sim com base em dados reais dos usuários finais.

Normalmente o termo agilidade é diretamente relacionado com processos de organização de trabalho e planejamento, e não necessariamente com o desenvolvimento de software. No entanto, como mencionado no critério ***Escalabilidade da aplicação*** da Seção 4.2.2, o sistema de software pode ser implementado de forma que novas funcionalidades sejam facilmente adicionadas, trazendo mais agilidade e reduzindo o *time-to-market*. Com esse direcionamento, muitas vezes a entrega rápida do software será priorizada sobre a resolução de *bugs* ou sobre a melhor experiência possível. Uma vez que as hipóteses tenham sido validadas, os desenvolvedores podem voltar seu foco para o polimento da

solução.

A escolha desse pilar tem o intuito de guiar a aplicação sempre para o caminho mais simples, ao invés da implementação de soluções grandes e complexas, melhorando a relação entre benefício e esforço. Os princípios de ***Implantação em nuvem*** e ***Implantação contínua*** do padrão *Twelve Factor Application* (Tabela 4.11) podem contribuir de forma significativa para o aumento da agilidade no desenvolvimento do sistema.

4.2.4 Baixo Custo

Muitos projetos, principalmente no contexto de *startups*, não possuem um grande financiamento para desenvolver as suas soluções. Por isso, esses projetos não podem arcar com grandes custos para implantação e manutenção de seus sistemas. Como mencionado nas seções anteriores, os desenvolvedores podem adotar estratégias para criar versões mais simples das soluções para que as hipóteses de negócio sejam validadas. No entanto, uma aplicação simples não é o suficiente para que o custo total de um projeto se torne baixo. Existem diversos outros custos envolvidos no planejamento, desenvolvimento, manutenção, monitoramento e, principalmente, implantação e hospedagem de um sistema.

Esse pilar visa orientar as decisões de projeto na direção de estratégias que minimizem o custo total envolvido na criação da solução. Como veremos adiante, a adoção de computação em nuvem tem um papel muito importante para manter as despesas envolvidas com a hospedagem do sistema baixas. Além disso, os pilares e princípios definidos anteriormente também são de suma importância para garantir que os custos de desenvolvimento e manutenção das aplicações continuem baixos.

4.2.5 Síntese

Com a definição de todos esses pilares, critérios e princípios, se torna mais claro o caminho para o desenvolvimento de um sistema que vá de encontro às prioridades do projeto como um todo. A necessidade é de um sistema simples, facilmente expansível, que possa ser implementado de forma evolutiva e que não traga grandes custos. Nas próximas seções, veremos os resultados da escolha desses pilares na arquitetura e desenvolvimento das aplicações.

4.3 Arquitetura Geral

Nessa parte do trabalho, entraremos em maiores detalhes na arquitetura da plataforma *Skate King*. O sistema é composto por duas aplicações clientes que se comunicam com uma aplicação servidor. Esse servidor, por sua vez, alterna suas comunicações entre dois bancos de dados diferentes, cada um com um propósito específico. Essa organização pode ser vista na Figura 4.2 e a implementação de cada componente é detalhada nas seções a seguir.

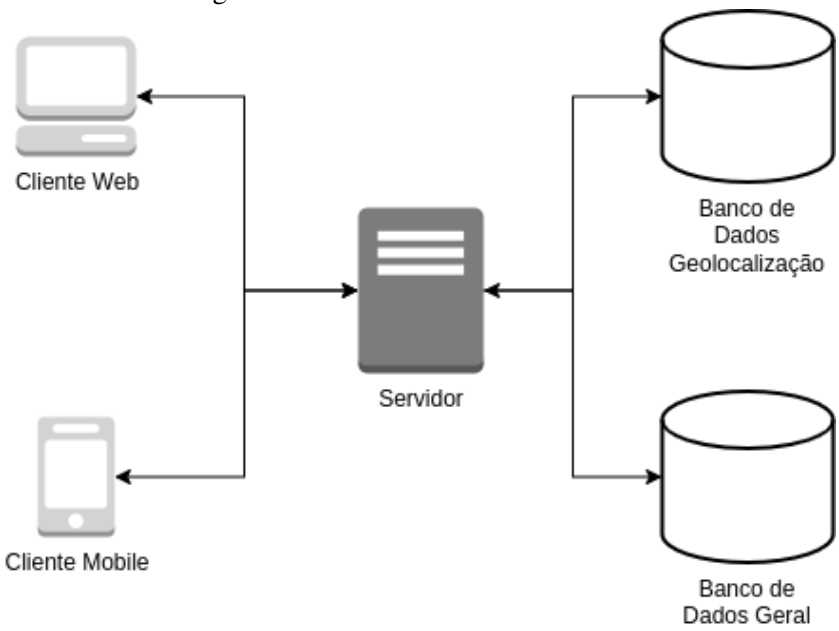
4.3.1 Cliente-Servidor

Como apresentado brevemente nas seções anteriores, a arquitetura alto nível do sistema foi baseada no modelo Cliente-Servidor. Nesse modelo, uma aplicação que executa no dispositivo *Cliente* envia requisições pela rede para outra aplicação que executa no *Servidor* e se comunica com o *Banco de Dados*. Esse modelo é utilizado por grande parte das aplicações de internet modernas. Isso porque, ao organizar o sistema dessa forma, facilita-se o desacoplamento entre o *front-end* e o *back-end*, possibilitando que sejam executados em ambientes totalmente segregados e de forma independente. Ao adotar tal organização para esse sistema, foi possível implementar duas aplicações *front-end*, uma para dispositivos móveis e outra para navegadores web, que utilizam um mesmo *back-end*.

Além disso, também foram utilizados dois bancos de dados que se comunicam com a aplicação servidor. Um dos bancos de dados é destinado para dados gerais da aplicação e o outro é estruturado e otimizado especificamente para leituras e escritas de dados de geolocalização. O servidor, quando necessário, replica os dados armazenados em um banco de dados para o outro. Da mesma forma, a depender do tipo de consulta que precisa ser feita, o servidor alterna suas leituras entre os dois bancos de dados.

Essa estrutura é ilustrada na Figura 4.2 a seguir.

Figura 4.2 – Cliente-Servidor detalhado



Fonte: Autor

4.3.2 *Front-end Mobile*

Esse componente do sistema se trata de uma aplicação para dispositivos móveis iOS e Android. Para essa implementação, foi utilizada a ferramenta híbrida *React Native*. Com isso, é possível ter apenas uma base de código que é compilada para aplicações nativas de ambas as plataformas. O código é implementado na linguagem *Javascript*, utilizando a sintaxe *JSX*, semelhante ao *HTML*. Essa biblioteca utiliza uma organização em componentes que podem ser utilizados em diversos locais da aplicação. Esses componentes são basicamente funções que retornam a configuração de um *layout* codificada utilizando a sintaxe declarativa *JSX*. Essas funções podem ser utilizadas em diversos contextos dentro do aplicativo, criando uma arquitetura modular e que facilita o reuso. A Figura 4.3 mostra um exemplo do código de um componente da aplicação.

Figura 4.3 – Exemplo de um componente da aplicação React Native

```
const CheckinButton = ({ status, onPress }) => {
  const renderButton = () => {
    switch (status) {
      case "success":
        return <CheckImage source={check} />;
      case "error":
        return <ErrorImage source={exclamation} />;

      case "loading":
        return <ActivityIndicator color="#474F5A" size={50} />;

      default:
        return <CheckinImage source={checkin} />;
    }
  };

  return (
    <Container onPress={onPress}>
      {renderButton()}
    </Container>
  );
};
```

Fonte: Autor

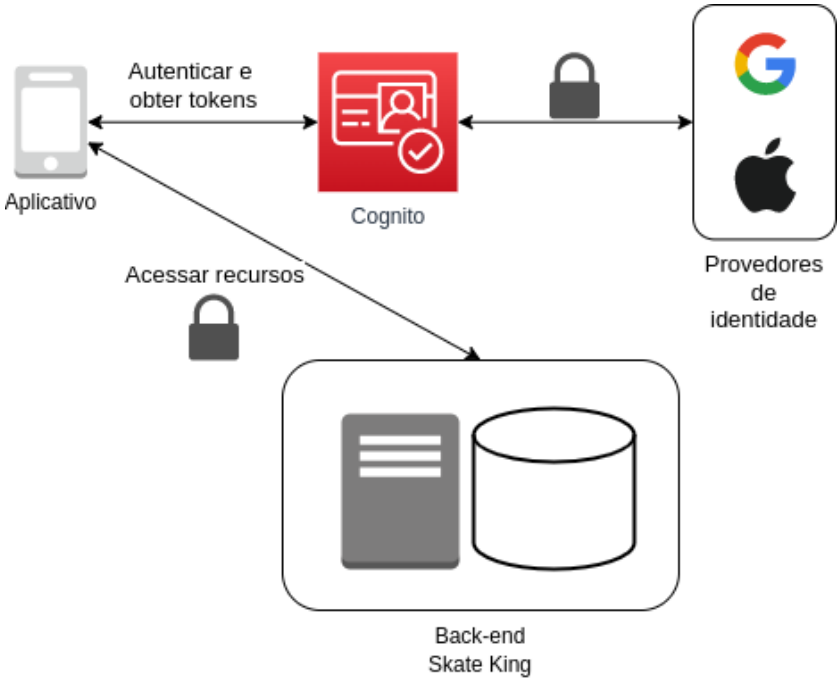
Nessa aplicação, os componentes foram separados entre *telas* e *componentes comuns*. Os componentes tela nada mais são do que componentes que ocupam toda a interface do dispositivo, representando uma página da aplicação. Os componentes comuns são diversos e normalmente são organizados de forma que possam ser reutilizados em vários locais do aplicativo. O exemplo da Figura 4.3 corresponde ao componente responsável por renderizar o botão da ação de *check-in*. Como pode ser visto no exemplo, componentes, por serem funções, podem receber parâmetros e definir o *layout* a ser renderizado com base em uma lógica executada internamente.

O aplicativo busca oferecer ao usuário uma experiência simples, permitindo que ele navegue pelo mapa, clique em *pins* para ver detalhes de *picos* e cadastre novos *picos*. Para isso, após realizar o login, o usuário é direcionado para a tela principal do aplicativo. Essa tela é a tela do mapa, onde são renderizados ícones na localização de cada *pico*. Ao clicar em um desses ícones, é exibido um *bottom-sheet* que pode ser expandido para revelar informações detalhadas sobre o *pico* como foto, descrição, número de *check-ins* e avaliações feitas por outros usuários. Além disso, existe um botão no canto inferior direito que, ao ser pressionado, leva a uma segunda tela onde o usuário tem as opções de

visualizar ou editar o seu perfil e cadastrar um novo pico. Ambas opções levam a outras telas onde os campos necessários podem ser preenchidos.

Para que seja possível realizar todas essas ações, o usuário deve primeiro fazer o cadastro e login. Essas operações são realizadas com uma conta do *Google* ou *Apple* já existente. Para a implementação desses fluxos de autenticação, foi utilizado um serviço do provedor de nuvem *AWS* chamado *Cognito*. Com esse serviço, o aplicativo tem a funcionalidade de *Single Sign-On*, permitindo que os usuários realizem cadastro e login sociais com provedores de identidade como Google e Apple. O serviço Cognito é responsável pela complexidade de lidar com os *tokens* que são retornados por esses provedores terceiros. Com isso, a aplicação recebe os *tokens* necessários diretamente do *Cognito* e pode usá-los para autenticar suas chamadas para o *back-end*. Essa dinâmica é ilustrada na Figura 4.4.

Figura 4.4 – Fluxo de autenticação entre *front-end*, *back-end* e *Cognito*



Fonte: Autor

4.3.3 *Front-end Web*

A implementação desse componente do sistema tem o intuito de prover aos mantenedores da plataforma um módulo gerencial com as funcionalidades necessárias para manter o sistema em pleno funcionamento. Por meio de uma interface Web, os usuários responsáveis pelo gerenciamento da plataforma devem ser capazes de analisar métricas

sobre o uso do aplicativo além de visualizar, cadastrar e deletar *picos*. Para essa implementação, foi usada a biblioteca *React.js*. Essa ferramenta é muito semelhante ao *React Native*, utilizado na implementação do *front-end mobile* (Seção 4.3.2), com algumas particularidades específicas de aplicações para navegadores web. Com isso, foi possível aproveitar os mesmos conhecimentos adquiridos no desenvolvimento do aplicativo para a implementação do sistema gerencial.

Da mesma forma que o *React Native*, o *React.js* permite que o código seja organizado em componentes, facilitando o reúso. A Figura 4.5 mostra um exemplo de um dos componentes do sistema, implementado com *React.js*. Além disso, para essa aplicação, foi utilizada a linguagem *Typescript*, que é um *superset* da linguagem *Javascript*. Com isso, é possível aproveitar os benefícios de linguagens estaticamente tipadas e, ao mesmo tempo, a liberdade de linguagens dinâmicas como o *Javascript*.

Figura 4.5 – Exemplo de um componente da aplicação React.js

```
export function Sidebar({ onPress }: SideBarProps) {
  return (
    <SideBarContainer>
      <SpotImage src={mapMarkerImg} alt="SkateKing" />

      <FooterContainer>
        <BackButton type="button" onClick={onPress}>
          <FiArrowLeft size={24} color="#232f3d" />
        </BackButton>
      </FooterContainer>
    </SideBarContainer>
  );
}
```

Fonte: Autor

Assim como no aplicativo móvel, a tela principal é a tela do mapa. Nessa tela, são exibidas algumas métricas sobre o aplicativo e é possível navegar com o mouse pelo mapa, onde ícones são renderizados na posição de cada *pico*. Ao clicar em um desses ícones, é exibida uma nova tela com as informações detalhadas do pico, da mesma forma que no aplicativo. No caso do sistema gerencial, é possível deletar um *pico* a partir da tela de detalhes. Além disso, na tela do mapa, há um botão no canto inferior direito que leva à pagina de criação de um *pico*, onde os campos necessários devem ser preenchidos.

4.3.4 *Back-end*

Grande parte da aplicação *back-end* do sistema consiste em uma API REST que se comunica com os dois bancos de dados. Essa API é composta por diversos *endpoints*, cada um deles representando uma ação e um recurso. A depender do caminho e do verbo HTTP utilizados em uma requisição, o servidor define qual ação será executada. Como apresentado anteriormente, um dos bancos de dados é destinado a operções gerais do sistema e o outro é otimizado especificamente para dados de geolocalização. Alguns *endpoints* fazem leituras e escritas em apenas um dos bancos de dados, enquanto outros fazem operações em ambos. A estrutura criada especificamente para dados de geolocalização será descrita em detalhes em outra seção. As Tabelas 4.14, 4.13, 4.15 e 4.16 a seguir mostram todos os *endpoints* (verbo e caminho) implementados na aplicação *back-end* e suas respectivas descrições. Todos os *endpoints* recebem, via *Headers*, um token que é usado pelo servidor para validar a autenticação do usuário no *Cognito*. O código foi implementado na linguagem *Typescript* e executado utilizando o *runtime Node.js*.

Além dos *endpoints*, também foram implementados mecanismos para replicar os dados de usuários cadastrados no *Cognito* para o banco de dados da aplicação. Dessa forma, é possível complementar os dados básicos de cadastro com informações do perfil do usuário específicas ao domínio da aplicação. Nas próximas Seções, são descritos mais detalhes sobre a arquitetura e implementação do *back-end*.

Tabela 4.13 – *Endpoints* da aplicação *back-end* relacionados à *check-ins*

Endpoint	***Descrição***
POST /spots/:id/checkins	Consulta o banco de dados de geolocalização para verificar se o usuário está próximo ao pico em questão. Se a posição for dentro de um raio de 500 metros em torno do *pico*, cria 2 registros de *check-in* no banco de dados geral, um temporário e um permanente. Recebe o identificador do *pico* via parâmetros de URL e os dados do *check-in* em formato JSON no corpo da requisição.
GET /metrics	Retorna contagem de usuários cadastrados, *picos* cadastrados e *check-ins* realizados.

Fonte: Autor

Tabela 4.14 – *Endpoints* da aplicação *back-end* relacionados à *picos*

Endpoint	***Descrição***
POST /spots	Cria um *pico* no banco de dados geral e no banco de dados de geolocalização. Recebe os campos necessários em formato *JSON* no corpo da requisição.
DELETE /spots/:id	Remove um *pico* dos dois bancos de dados. Recebe o identificador do *pico* via parâmetros de URL.
GET /spots/:id	Retorna dados completos sobre um *pico*. Recebe o identificador do *pico* via parâmetros de URL. Retorna, em formato JSON, os campos referentes à identificador, endereço, cidade, categoria, dados do criador, descrição, URL da foto, latitude, longitude, nome, tags, telefone de contato (caso seja uma loja), número de *check-ins* recentes e número total de *check-ins*.
GET /spots	Retorna uma listagem de picos próximos a um ponto informado. Recebe os dados de latitude e longitude do ponto central, além do valor do raio, via parâmetros de *querystring*. Essa consulta é feita no banco de dados otimizado para geolocalização.

Fonte: Autor

Tabela 4.15 – *Endpoints* da aplicação *back-end* relacionados à avaliações

Endpoint	***Descrição***
POST /spots/:id/ratings	Cria uma avalição para um *pico* no banco de dados geral. Recebe o identificador do *pico* via parâmetros de URL e os dados na avaliação em formato JSON no corpo da requisição.
GET /spots/:id/ratings	Retorna uma listagem das avaliações de um *pico*. Recebe o identificador do *pico* via parâmetros de URL. Para cada avaliação, retorna, em formato JSON, campos referentes à nota, comentário, data de criação e dados do criador.

Fonte: Autor

Tabela 4.16 – *Endpoints* da aplicação *back-end* relacionados à usuários

Endpoint	***Descrição***
HEAD /users/:username	Verifica se existe um usuário com o nome de usuário fornecido. Retorna a existência ou não do usuário via HTTP *Status Code*.
GET /users	Retorna uma listagem do usuários cadastrados. Recebe um token de paginação que define qual página da listagem será retornada.
PATCH /users/me	Edita os dados do perfil do usuário.
GET /users/:id	Retorna dados do perfil de um usuário. Recebe o identificador do usuário via parâmetros de URL

Fonte: Autor

4.4 Arquitetura do *Back-end*

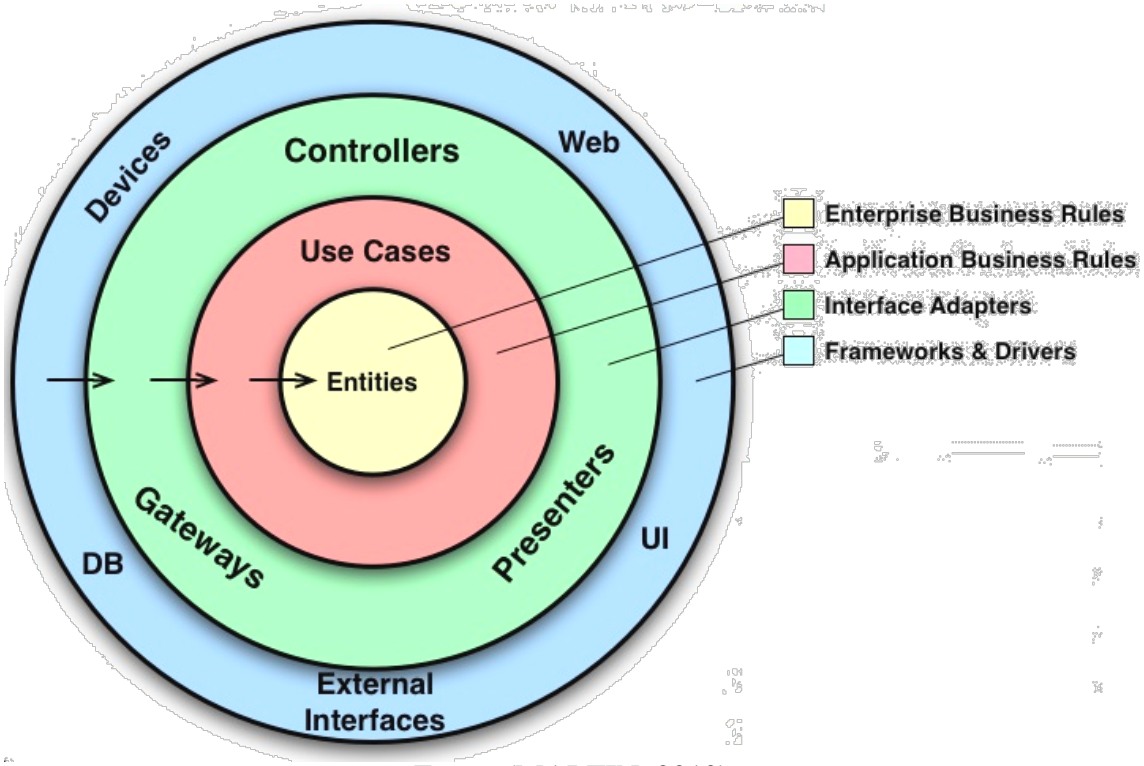
A aplicação *back-end* foi organizada pensando principalmente nos pilares detalhados anteriormente. Para a definição da arquitetura, serviram como base os padrões de desenvolvimento mencionados na seção 4.2.1. Nessa seção, será apresentada a arquitetura e *design* do sistema por dois pontos de vista: *aplicação* e *infraestrutura*. Na visão de *aplicação*, voltaremos o foco da análise para a organização do código e dos componentes lógicos do sistema. Já na visão de *infraestrutura*, detalharemos a organização dos recursos de infraestrutura, com foco na computação em nuvem.

4.4.1 Arquitetura da Aplicação

O padrão de arquitetura *Clean Architecture* prega a separação do software em camadas. Essa separação tem como principal objetivo a segregação de responsabilidades da aplicação. Cada camada tem um papel específico no funcionamento do sistema como um todo. A hierarquia de componentes do sistema é organizada em um formato de "cebola". No centro estão os componentes principais para o domínio da aplicação. Esses componentes são os que sofrerão menos mudanças ao longo do tempo, já que representam a lógica do que é o diferencial do negócio. Nas camadas mais externas estão os componentes menos importantes para o domínio do negócio e que poderiam facilmente ser substituídos por versões alternativas.

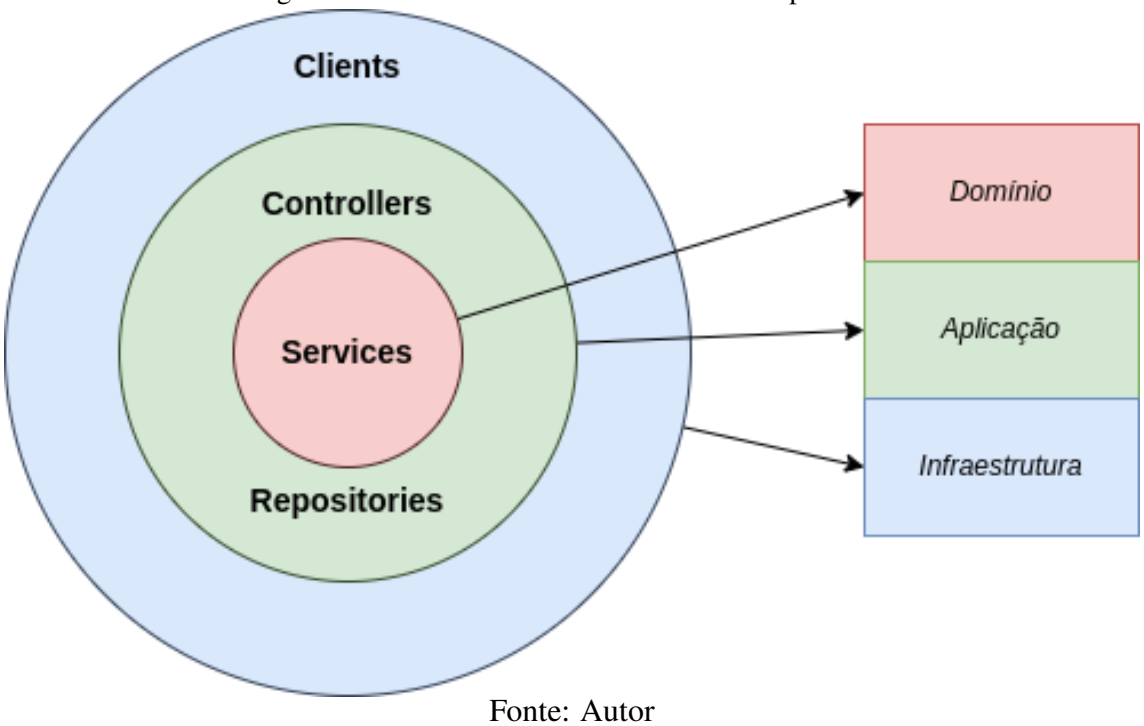
Dessa forma, foi adotada, para esse sistema, uma separação da aplicação em camadas. No entanto, como mencionado na Seção 4.2, os pilares que orientam o *design* do sistema prezam por uma estrutura simples e de fácil implementação. Portanto, foi utilizada uma versão simplificada da hierarquia de camadas proposta pelo padrão *Clean Architecture*. A Figura 4.6 abaixo mostra a separação em camadas do *Clean Architecture* e, em seguida, a Figura 4.7 apresenta a versão simplificada que foi implementada nesse trabalho.

Figura 4.6 – Camadas do padrão *Clean Architecture*



Fonte: (MARTIN, 2012)

Figura 4.7 – Camadas *Clean Architecture* Simplificado



Fonte: Autor

Como pode ser visto na Figura 4.7, a aplicação foi organizada entre três camadas: *Domínio*, *Aplicação* e *Infraestrutura*. Cada uma das camadas tem suas responsabilidades bem definidas e o acoplamento entre as camadas é mínimo.

A camada de *infraestrutura* é composta por todos os componentes externos ao sistema, como banco de dados e bibliotecas ou *plugins* terceiros. Nesse sistema a camada de infraestrutura foi utilizada apenas para centralizar a comunicação com esses componentes externos, a fim de minimizar o acoplamento e tornar possível a substituição de uma ferramenta ou tecnologia sem causar impactos no restante do sistema. Os componentes dessa camada tem como função garantir que os princípios *Independência de Frameworks* e *Independência do Banco de Dados* (Tabela 4.10) sejam respeitados.

Os componentes da camada de *aplicação* são responsáveis pela adaptação dos dados para comunicação entre as camadas de domínio e infraestrutura. Com isso, a lógica de domínio é isolada dos diferentes pontos de entrada (*entrypoints*) e repositórios de dados (*dataproviders*) do sistema. No caso dessa aplicação, os principais componentes dessa camada são os *controllers*, responsáveis por adaptar os dados na comunicação dos *entrypoints* com o domínio, e os *repositories*, responsáveis por adaptar os dados enviados pelo domínio para um *dataprovider* externo. Esses componentes serão descritos em maior detalhe nas seções seguintes. Com essa camada, são reforçados os princípios de *Testabilidade* (Tabela 4.10) e *Foco no Domínio* (Tabela 4.12).

Por fim, a camada de domínio é o "coração"do sistema, contendo as regras de negócio que trazem valor para o usuário final. Nessa camada estão implementados os *use-cases* da aplicação (nesse caso chamados de *services*). Como esses componentes foram implementados de forma isolada, é possível que sejam testados de forma independente e não são afetados por mudanças em ferramentas ou tecnologias utilizadas. Além disso, se o princípio de *Complexidade do Domínio* for respeitado, a maior parte da complexidade do sistema deve estar localizada nos componentes dessa camada.

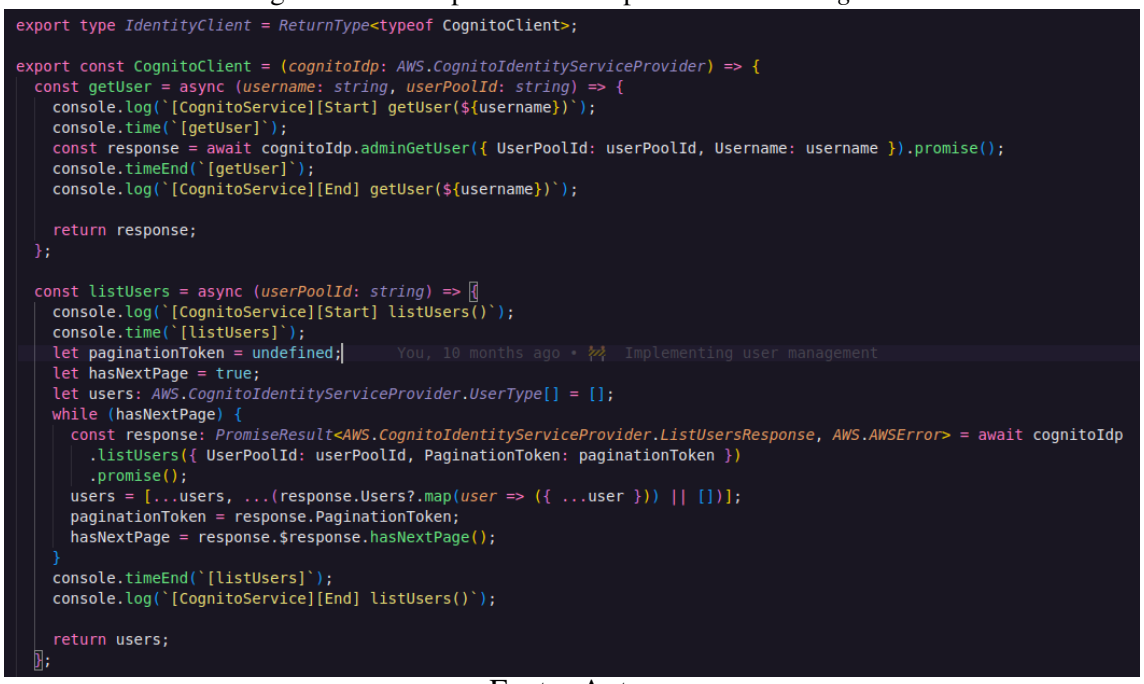
A seguir, será descrito cada tipo de componente implementado e qual o seu papel no funcionamento do sistema.

4.4.1.1 Clients

Esses componentes fazem parte da camada de *infraestrutura*. Para cada biblioteca terceira, foi criado um componente *client* responsável por centralizar toda a interação com essa biblioteca. Dessa forma, os componentes que precisam utilizar funcionalidades de uma biblioteca externa ficam acoplados a um componente interno do sistema. Esses componentes, caso necessário, podem ser adaptados para utilizar outra biblioteca internamente, mantendo o restante da aplicação em funcionamento. A Figura 4.8 mostra um trecho de código do *client* criado para centralizar a comunicação com o SDK que realiza

operações no serviço *Cognito*.

Figura 4.8 – Componente *client* para o SDK do *Cognito*



```
export type IdentityClient = ReturnType<typeof CognitoClient>;

export const CognitoClient = (cognitoIdp: AWS.CognitoIdentityServiceProvider) => {
  const getUser = async (username: string, userPoolId: string) => {
    console.log(`[CognitoService][Start] getUser(${username})`);
    console.time(`[getUser]`);
    const response = await cognitoIdp.adminGetUser({ UserPoolId: userPoolId, Username: username }).promise();
    console.timeEnd(`[getUser]`);
    console.log(`[CognitoService][End] getUser(${username})`);

    return response;
  };

  const listUsers = async (userPoolId: string) => {
    console.log(`[CognitoService][Start] listUsers()`);
    console.time(`[listUsers]`);
    let paginationToken = undefined;      You, 10 months ago • 🚧 Implementing user management
    let hasNextPage = true;
    let users: AWS.CognitoIdentityServiceProvider.UserType[] = [];
    while (hasNextPage) {
      const response: PromiseResult<AWS.CognitoIdentityServiceProvider.ListUsersResponse, AWS.AWSError> = await cognitoIdp
        .listUsers({ UserPoolId: userPoolId, PaginationToken: paginationToken })
        .promise();
      users = [...users, ...(response.Users?.map(user => ({ ...user })) || [])];
      paginationToken = response.PaginationToken;
      hasNextPage = response.$response.hasNextPage();
    }
    console.timeEnd(`[listUsers]`);
    console.log(`[CognitoService][End] listUsers()`);

    return users;
  };
```

Fonte: Autor

4.4.1.2 Controllers

Esses componentes fazem parte da camada de *aplicação*. Como descrito na Seção 4.4.1, os componentes *controller* são responsáveis por adaptar a comunicação de um *entrypoint* com a camada de domínio. Um *entrypoint* pode ser qualquer ponto de entrada da aplicação que recebe mensagens de fora do sistema. Alguns exemplos de *entrypoints* que podem ser citados são: sistemas de fila e mensageria, *endpoints* que recebem requisições HTTP, operações no sistema via linha de comando (CLI), entre outros. Em qualquer um desses casos, um *controller* seria responsável por adaptar a entrada para um formato padrão reconhecido pela camada de domínio e adaptar o retorno da camada de domínio para o *entrypoint* em questão. Nessa aplicação, grande parte dos *controllers* gerenciam a interação da camada de domínio com requisições recebidas em *endpoints* HTTP. A Figura 4.9 contém a função principal do *controller* responsável pelo *endpoint* de criação de *check-in*.

Figura 4.9 – Componente *controller* do *endpoint* de criação de *check-in*

```
const handler = async (event: APIGatewayEvent) => {
  try {
    const inputSpot = validate({
      ...JSON.parse(event.body ?? '{}'),
      spotId: event.pathParameters?.['id'],
    });

    const data = await service.execute(inputSpot);

    return {
      statusCode: Status.CREATED,
      headers: corsHeaders,
      body: JSON.stringify({
        success: true,
        message: 'Check-in created successfully',
        data,
      }),
    };
  } catch (err) {
    console.error(err);
    return {
      statusCode: Status.INTERNAL_SERVER_ERROR,
      body: JSON.stringify({
        succes: false,
        message: (err as Error).message || err,
      }),
    };
  }
};
```

Fonte: Autor

4.4.1.3 Repositories

Assim como os componentes *controller*, os *repositories* também estão localizados na camada de *aplicação*. Dessa forma, suas responsabilidades são bastante semelhantes com as dos *controllers*. Esses componentes centralizam a interação da camada de domínio com sistemas provedores de dados. Esses provedores de dados podem ser qualquer sistema que lide com armazenamento e leitura de informações, como bancos de dados, sistemas de arquivos, APIs REST externas ou até mesmo um simples armazenamento em memória. No caso desse sistema, todos os repositories foram implementados para gerenciarem a comunicação com o banco de dados *DynamoDB*. A Figura 4.10 apresenta o código do *repository* responsável pelas operações de criação de um *pico* nos dois bancos de dados da aplicação.

Figura 4.10 – Componente *repository* para criação de um *pico*

```
export function CreateSpotRepositoryFactory(geoDataClient: GeoDataClient, databaseClient: DatabaseClient) {
  const createGeoSpot = async (leanSpot: LeanSpot) => {
    const response = await geoDataClient.putPoint({
      id: leanSpot.id,
      point: { latitude: leanSpot.latitude, longitude: leanSpot.longitude },
      attributes: { ...leanSpot },
    });

    return response;
  };

  const createSpot = async (spot: Spot) => {
    const response = await databaseClient.put({
      TableName: config.dataTableName,
      Item: {
        ...spot,
        partition_key: `entity#${DatabaseEntities.spot}`,
        sort_key: `${DatabaseEntities.spot}#${spot.id}`,
        entity: DatabaseEntities.spot,
      },
    });

    return response;
  };

  return { createGeoSpot, createSpot };
}
```

Fonte: Autor

4.4.1.4 Services

Os componentes *service* estão localizados na camada de domínio. Dessa forma, são responsáveis pela lógica central de negócio. Um componente *service* normalmente utiliza componentes *repository* para ler e armazenar dados. Essa porção de código deve ser testada isoladamente para garantir que as regras do domínio estão sendo respeitadas. Como descrito nos princípios de *Foco no Domínio* e *Complexidade do Domínio* (Tabela 4.12), os componentes *service* costumam conter a maior quantidade de código e a maior complexidade do sistema. A Figura 4.11 exibe um trecho de código do *service* responsável pela criação de uma avaliação.

Figura 4.11 – Componente *service* para criação de uma avaliação

```
const execute = async (inputSpotRating: InputSpotRating) => {
  const userId = generateUuidFromString(inputSpotRating.userEmail);

  const creator = (await repository.getUser(userId)) as DatabaseUser;

  if (!creator) {
    throw new Error('Could not find user');
  }

  const now = new Date();

  const spotRating: SpotRating = {
    id: uuid(),
    createdAt: now.toISOString(),
    spotId: inputSpotRating.spotId,
    userId,
    rating: inputSpotRating.rating,
    comment: inputSpotRating.comment,
    creatorName: creator?.name,
    creatorPicture: creator?.picture,
    creatorUsername: creator?.username,
  };

  await repository.createSpotRating(spotRating);

  return { id: spotRating.id };
};
```

Fonte: Autor

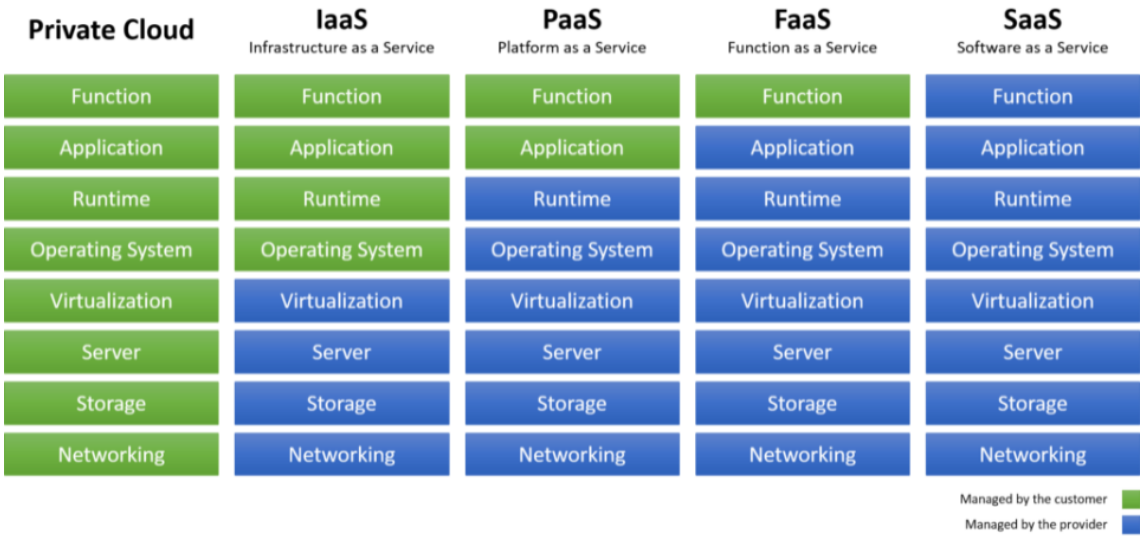
4.4.2 Arquitetura da Infraestrutura

Considerando, principalmente, o pilar de *Escalabilidade* definido na Seção 4.2.2, foi feita a escolha de hospedar a aplicação em um ambiente de nuvem. Dessa forma, se torna mais simples escalar os recursos alocados ao sistema conforme o volume aumenta. Para isso, como mencionado de forma breve nas seções anteriores, foi escolhida a nuvem *AWS*, uma das principais provedoras de computação em nuvem do mercado. Ao se adotar uma plataforma em nuvem para hospedar a aplicação, algumas responsabilidades são "transferidas"para o provedor, como a manutenção e proteção dos servidores físicos responsáveis pela hospedagem.

Dentro do modelo de computação em nuvem, existem diferentes níveis de abstração sobre a infraestrutura utilizada. Um desenvolvedor pode alugar uma máquina virtual e instalar e expor a sua aplicação para internet ou então utilizar um serviço que geren-

cia automaticamente um *cluster* de servidores com diversas instâncias e balanceamento de carga. Essa divisão de responsabilidades entre cliente e provedor em cada nível de abstração é ilustrada no diagrama da Figura 4.12. Mais à esquerda estão os modelos de nuvem privada, onde o cliente tem a própria infraestrutura e assume todas as responsabilidades. À direita está o modelo de *Software as a Service* (ou *SaaS*), onde o provedor abstrai grande parte da lógica e gerenciamento envolvida com o serviço em questão. O serviço *Cognito* mencionado anteriormente é um exemplo de serviço do modelo *SaaS* para controle de identidade e acesso de usuários.

Figura 4.12 – Devisão de responsabilidades entre provedor e cliente



Fonte: AWS

No contexto de computação em nuvem, um serviço totalmente gerenciado é um serviço fornecido por um provedor de nuvem que oferece um conjunto de funcionalidades específico cujas preocupações operacionais, como provisionamento, aplicação de *patches* e dimensionamento, são gerenciadas pelo fornecedor de nuvem e abstraídas do desenvolvedor (SWAIL, 2019). Dessa forma, todo esforço necessário para escalar e manter a infraestrutura passa a ser responsabilidade do provedor, ficando a cargo do desenvolvedor apenas o que é realmente importante para a sua aplicação.

A partir desse tipo de serviço, surgiu o conceito de computação *Serverless*. Computação *Serverless* permite que os desenvolvedores construam e executem aplicações e serviços sem pensar em servidores. Aplicações *Serverless* não exigem provisionamento, dimensionamento ou gerenciamento de servidores. Esse tipo e aplicação pode ser construído para quase qualquer contexto, e tudo que é necessário para executar e escalar a

aplicação com alta disponibilidade é gerenciado pelo provedor. Além disso, em aplicações *Serverless*, não há custo por capacidade não utilizada. Dessa forma, não é necessário provisionar a capacidade necessária pensando no pico de utilização e pagar por essa capacidade durante os períodos de baixo volume. Toda a logística relacionada com o provisionamento de recursos durante momentos de alta utilização, e com o desligamento de recursos quando não há utilização, é de responsabilidade do provedor.

Tendo em vista os pilares de *Escalabilidade* (Seção 4.2.2), *Agilidade* (Seção 4.2.3) e *Baixo Custo* (Seção 4.2.4), e os princípios de *Foco no Domínio*, *Complexidade do Domínio* (Tabela 4.12) e *Implantação em Nuvem* (Tabela 4.11), fica muito claro que a criação de uma arquitetura *Serverless* para esse sistema é de grande valia. Foram escolhidos 5 principais serviços totalmente gerenciados da nuvem AWS para a infraestrutura dessa aplicação. Esses serviços são descritos na Tabela 4.17. A Figura 4.13 mostra o diagrama da organização desses serviços para compor o sistema.

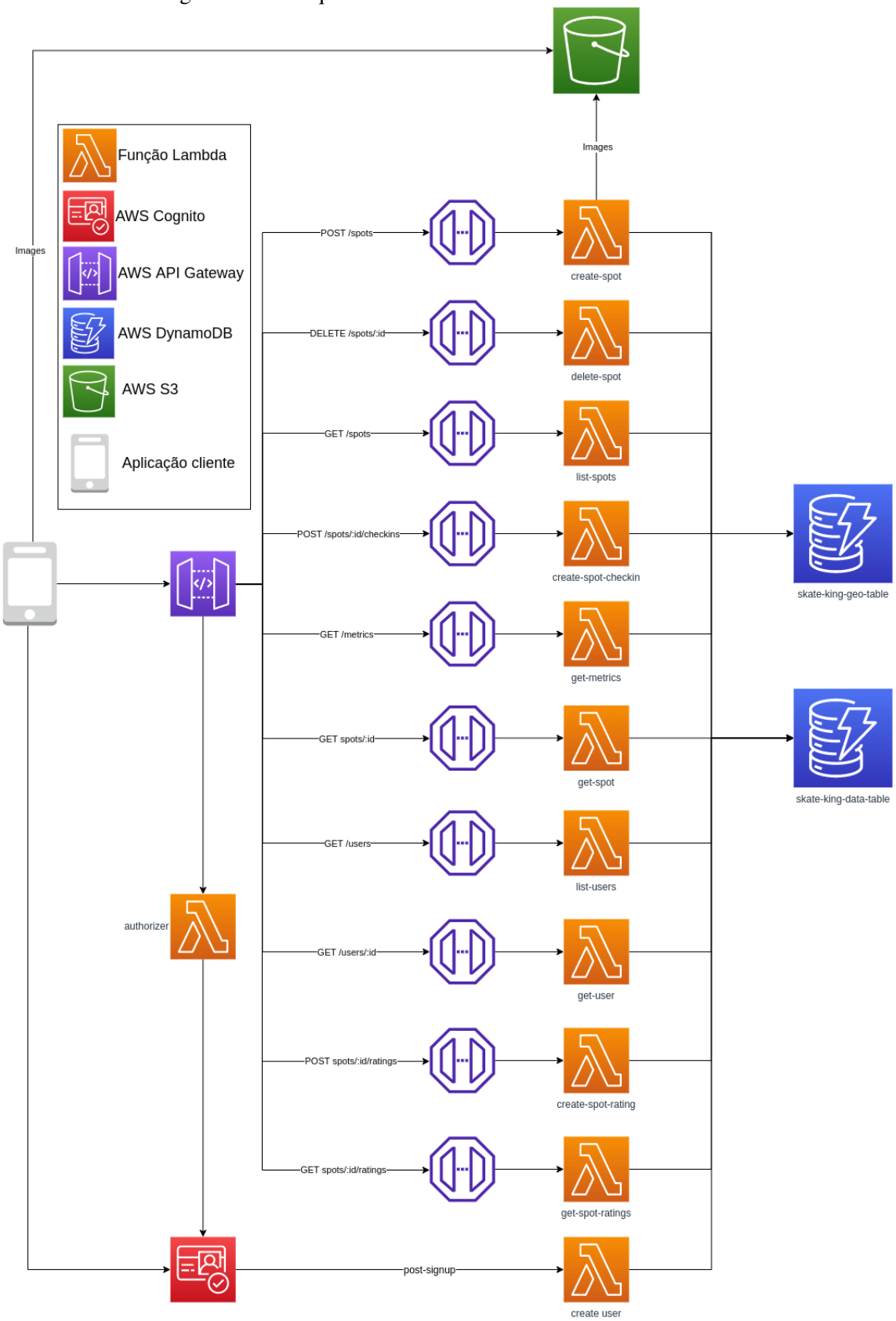
O AWS Lambda é um serviço do modelo *Functions as a Service* (FaaS), em que o desenvolvedor só é responsável pelo upload do código de uma função, assim como a configuração de um evento que deve disparar a execução dessa função. Como ilustrado na Figura 4.13, o código de cada *endpoint* da API REST foi hospedado em uma função *Lambda* dedicada. Dessa forma, a infraestrutura de um *endpoint* é totalmente segrada dos demais e problemas em uma função *Lambda* não têm nenhum impacto no restante do sistema. Para que as funções fossem acessíveis por meio de requisições HTTP, foi utilizado o serviço *API Gateway* para servir como "porta de entrada"da aplicação e criar um *endpoint* para cada uma das funções. Além disso, foi utilizada uma função *authorizer* para validar os tokens de acesso das requisições no serviço *Cognito*. O *Cognito*, por sua vez, foi utilizado para cadastro e *login* dos usuários diretamente do aplicativo. Após a finalização do cadastro de um usuário no *Cognito*, é disparado um gatilho que invoca uma função *Lambda*. Essa função é responsável por criar um registro para o usuário no banco de dados da aplicação. Como mencionado anteriormente, foram utilizados 2 tabelas *DynamoDB*, uma para dados em geral e outra otimizada para o armazenamento e consulta de dados de geolocalização. Por fim, o serviço *Simple Storage Service* (S3) serviu para o armazenamento de arquivos estáticos. Foram armazenados no S3 os arquivos de imagens postadas pelos usuários, assim como os arquivos HTML estáticos da página *web* do módulo gerencial.

Tabela 4.17 – Principais serviços da AWS utilizados

Serviço	***Descrição***
Api Gateway	Serviço gerenciado que permite que desenvolvedores criem, publiquem, mantenham, monitorem e protejam APIs em qualquer escala com facilidade. APIs agem como a "porta de entrada" para aplicativos acessarem dados, lógica de negócios ou funcionalidade de serviços de *back-end*.
Lambda	Serviço de computação *Serverless* e orientado a eventos que permite executar código para praticamente qualquer tipo de aplicação ou serviço de *back-end* sem provisionar ou gerenciar servidores.
DynamoDB	Banco de dados de chave-valor NoSQL, *Serverless* e totalmente gerenciado, projetado para executar aplicações de alta performance em qualquer escala.
Simple Storage Service (S3)	Serviço de armazenamento de objetos que oferece escalabilidade, disponibilidade de dados, segurança e performance líderes do setor. Permite armazenar e proteger qualquer quantidade de dados de praticamente qualquer caso de uso, como data lakes, aplicações nativas da nuvem e aplicações móveis.
Cognito	Permite adicionar cadastramento, login e controle de acesso de usuários a aplicações Web e móveis com rapidez e facilidade. Pode ser escalado para milhões de usuários e oferece suporte a login com provedores de identidade social como Apple, Facebook, Google e Amazon e com provedores de identidade empresariais via SAML 2.0 e OpenID Connect.

Fonte: AWS

Figura 4.13 – Arquitetura da infraestrutura do sistema na AWS



Fonte: Autor

4.5 Dados da Aplicação

Nessa seção, serão detalhados os dados que foram armazenados no banco de dados da aplicação. Será descrita a modelagem, assim como os tipos de consultas que precisaram ser atendidos. Além disso, serão apresentados os desafios impostos, e a forma como foram superados, ao trabalhar com dados de geolocalização.

4.5.1 Banco de dados

O banco de dados escolhido para aplicação foi o *Amazon DynamoDB*. Por se tratar de um banco de dados *NoSQL* e chave-valor, existem diversas particularidades que devem ser observadas durante a modelagem dos dados. No *DynamoDB*, um banco de dados é chamado de **tabela**. Cada tabela é composta por **itens**, e cada item é composto por **atributos**. Um atributo corresponde a um tipo de dado primitivo, de forma análoga ao conceito de uma *coluna* em um banco de dados relacional. Apesar de os itens de uma tabela não precisarem seguir um *schema* específico, alguns atributos são obrigatórios para todos os itens. Em uma tabela, um atributo precisa ser definido como **chave de partição** (também chamada de *hash key*) e, opcionalmente, outro atributo pode ser definido como **chave de ordenação** (ou *range key*). A combinação da chave de partição e chave de ordenação (se houver) de um item deve ser única dentro de uma tabela. A chave de partição é usada internamente pelo *DynamoDB* para calcular a partição onde um item será armazenado. A chave de ordenação é utilizada pelo *DynamoDB* para ordenar os itens de uma mesma partição.

Existem dois principais tipos de consultas que podem ser feitas em uma tabela do *DynamoDB*: *queries* e *scans*. A operação de *scan*, como o nome indica, percorre todos os itens armazenados na tabela e retorna apenas os itens que se encaixem com o filtro fornecido. Dessa forma, operações de *scan* são pouco eficientes e normalmente devem ser evitadas. As operações de *query*, por outro lado, utilizam o componente chave-valor do *DynamoDB* para fazer uma consulta direta em uma partição. No entanto, para que uma *query* possa ser executada, é necessário fornecer o valor da *hash key* para a partição em questão. Essa particularidade torna o processo de modelagem e escolha das chaves muito importante, já que uma escolha incorreta de chaves pode tornar as consultas no banco de dados mais custosa. Com isso, o processo de modelagem normalmente se inicia com a listagem dos padrões de consulta da aplicação. Na Tabela 4.18 são enumeradas as princi-

pais consultas para essa aplicação. Para casos onde novos padrões de consulta surgem ao longo do ciclo de vida da aplicação, podem ser criados **índices**, que são essencialmente réplicas da tabela original com uma escolha diferentes de chaves.

Tabela 4.18 – Principais padrões de consulta da aplicação

N.º	Consulta
1	Listar todos usuários
2	Buscar dados de um usuário a partir do seu *username*
3	Buscar dados de um usuário a partir do seu identificador
4	Listar *picos* dentro de um raio especificado a partir de um ponto
5	Listar avaliações de um *pico* a partir do seu identificador
6	Buscar dados de um pico a partir do seu identificador
7	Contar número de *check-ins* recentes de um pico a partir do seu identificador
8	Contar número de *check-ins* totais de um pico a partir do seu identificador
9	Contar número de *picos*
10	Contar número de *check-ins*
11	Contar número de usuários

Fonte: Autor

A partir dessa lista de padrões de consulta, foi criada a modelagem de chaves de partição e chaves de ordenação para cada entidade identificada. Essa modelagem é descrita na Tabela 4.19. São detalhados apenas os atributos de chave de partição e chave de ordenação. Os demais atributos são específicos de cada entidade e não são relevantes para a modelagem.

Os registros de *Check-in* Temporário foram utilizados em combinação com o mecanismo de *Time-To-Live* do *DynamoDB* para possibilitar a consulta número 7. Dessa forma, quando um usuário realiza *check-in* em um *pico*, são criados 2 itens - um temporário e um permanente. O registro temporário especifica um atributo chamado *expires_at*, que define um *timestamp* de quando esse item deve ser deletado automaticamente. Assim, definindo o valor desse atributo com um *timestamp* uma hora além da data atual, é possível consultar a contagem desse tipo de registro para determinar o número de *check-ins* em um *pico* dentro da última hora.

Tabela 4.19 – Escolha das chaves para cada entidade

Entidade	*partition_key*	*sort_key*
Usuário	*entity#user*	*user#USER_ID* / *USERNAME* (*index*)
Pico	*entity#spot*	*spot#SPOT_ID*
Avaliação	*spot#SPOT_ID*	*spot-rating#RATING_ID*
Check-in Temporário	*spot#SPOT_ID*	*temporary-spot-checkin#CHECKIN_ID*
Check-in Permanente	*spot#SPOT_ID*	*spot-checkin#CHECKIN_ID*

Fonte: Autor

O método de se representar diversas entidades na mesma tabela com o uso de prefixos (*user#*, *spot#*, etc.) se chama *Single Table Design*. Essa modelagem possibilita que grande parte dos padrões de consulta sejam atendidos com operações do tipo *query*. A Tabela 4.20 descreve, de forma simplificada, o plano para cada uma das consultas. No entanto, a consulta número 4, que depende de cálculos de geolocalização, ainda não pode ser facilmente realizada com apenas essa escolha de chaves. Por esse motivo foi criada uma segunda tabela *DynamoDB* especificamente para esse tipo de consulta. A estrutura dessa tabela é detalhada na próxima seção.

Tabela 4.20 – Plano de *query* para cada padrão de consulta identificado

N.º	Operação
1	*query(partition_key:=entity#user and begins_with(sort_key, user))*
2	*query(partition_key:=entity#user and username:=$USERNAME))*
3	*query(partition_key:=entity#user and sort_key:=user#$USER_ID))*
4	*N/A*
5	*query(partition_key:=spot#$SPOT_ID and begins_with(sort_key, spot-rating))*
6	*query(partition_key:=entity#spot and sort_key:=spot#$SPOT_ID))*
7	*query(partition_key:=spot#$SPOT_ID and begins_with(sort_key, temporary-spot-checkin))*
8	*query(partition_key:=spot#$SPOT_ID and begins_with(sort_key, spot-checkin))*
9	*query(partition_key:=entity#spot)*
10	*query(partition_key:=entity#spot-checkin)*
11	*query(partition_key:=entity#user)*

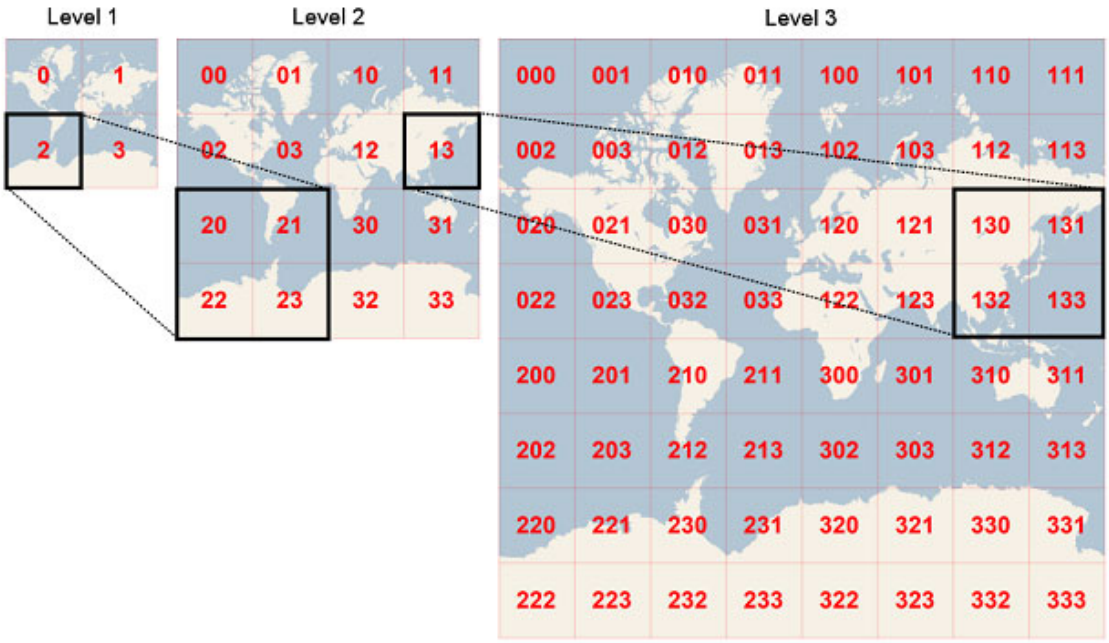
Fonte: Autor

4.5.2 Dados de Geolocalização

Consultas baseadas em geolocalização não são facilmente realizadas considerando as particularidades do *DynamoDB* com relação a escolha das chaves. Dessa forma, surge o principal desafio para o armazenamento de dados da aplicação: como modelar os registros de *picos* de forma que seja possível realizar consultas eficientes baseadas em geolocalização? A solução encontrada para esse problema foi a utilização de algoritmos de ***geohashing***.

Geohashing é uma técnica que subdivide a área do planeta em diversas regiões menores, cada uma representada por uma sequência de caracteres. O tamanho da área representada por cada código está relacionado com o número de caracteres dessa sequência. Se for utilizado apenas 1 caractere, serão representadas poucas regiões com áreas bastante grandes. Conforme o comprimento dessa chave de *geohash* aumenta, mais precisa se torna a representação. Da mesma forma, podem ser utilizados prefixos da chave para identificar regiões maiores dentro de uma representação mais precisa. Todas as subdivisões de uma região maior terão como prefixo a chave que representa a região maior. A Figura 4.14 ilustra os diferentes níveis de precisão em uma representação de regiões do planeta utilizando a técnica de *geohashing*.

Figura 4.14 – Divisões e subdivisões de regiões em uma representação de *geohash*



Fonte: https://www.pubnub.com/learn/glossary/what-is-geohashing/

Como pode ser visto na Figura 4.14, ao adicionar mais caraceteres à representa-

ção são criadas novas subdivisões em cada região. Com isso, pode-se usar um código de representação de *geohash* como chave de partição dos itens da tabela dedicada à dados de geolocalização. Assim, uma operação *query* pode ser executada para consultar *picos* que estão dentro de uma região representada por um código *geohash*. No entanto, ainda é necessário pensar nos padrões de consulta da aplicação, já que, dependendo do comprimento de chave *geohash* escolhido, as consultas podem se tornar mais ou menos eficientes com base no tamanho de **raio** utilizado nas consultas. Uma chave muito longa, com uma representação mais precisa, pode necessitar de diversas consultas para cobrir um valor de raio grande. Uma chave mais curta - e menos precisa - pode tornar as consultas menos eficientes, já que, se o valor do raio for pequeno, serão retornados mais dados do que o necessário. Para essa aplicação foi escolhido como chave de partição um prefixo de 6 dígitos da chave *geohash*. Esse comprimento é ideal para consultas com valores de raio em torno de 5 quilômetros.

A seguir são descritos, em alto nível, os passos necessários durante o processamento de uma consulta baseada em geolocalização:

1. Requisição HTTP no *endpoint GET /spots* informando valores de latitude e longitude do ponto central e um valor de raio;
2. Valores de latitude e longitude são convertidos em um código *geohash*;
3. É extraído o prefixo de 6 dígitos do código calculado;
4. Dependendo do valor do raio, são calculadas as células vizinhas do código *geohash* gerado;
5. São realizadas operações de *query* para o código *geohash* principal, assim como para os vizinhos;
6. Dados dos *picos* são retornados na *response* HTTP.

5 DEMONSTRAÇÃO

Esse Capítulo descreve em maiores detalhes a experiência do usuário em cada uma das funcionalidades descritas no Capítulo 4. O Capítulo está organizado de forma orientada pelas telas presentes durante os fluxos, apresentando os *layouts* e as interações possíveis em cada uma. Primeiramente, é demonstrado o funcionamento do aplicativo e, em seguida, o funcionamento do módulo gerencial.

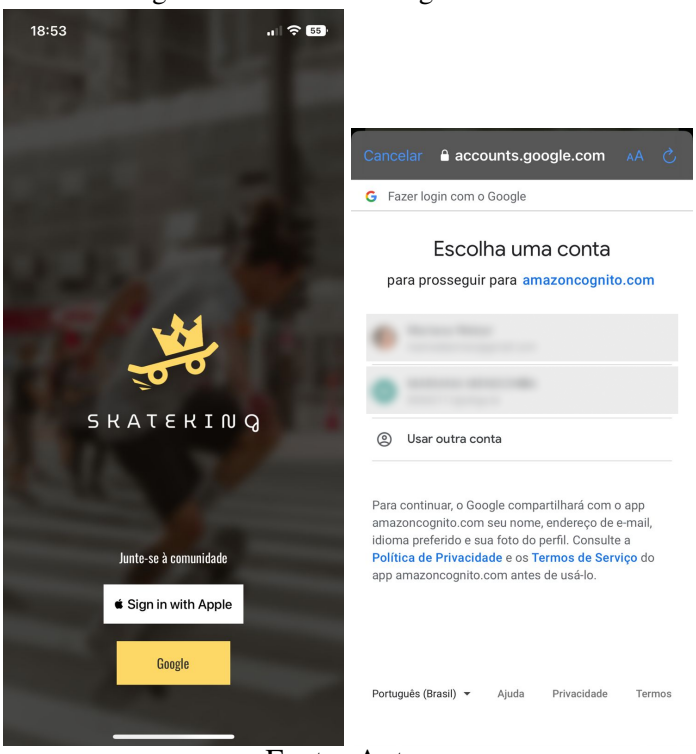
5.1 Funcionamento Aplicativo

Cada uma das seções a seguir corresponde a uma etapa da experiência do usuário durante o uso do aplicativo.

5.1.1 Tela de Login/Cadastro

Essa é a primeira tela apresentada ao usuário quando o aplicativo é aberto. Nessa tela, são exibidas as opções de realizar *login* ou cadastro utilizando contas de Google ou Apple. Essa tela também é apresentada caso o usuário escolha a opção de *logout* no aplicativo ou caso precise refazer o *login* quando sua seção é expirada. A interface é limpa e simples, é exibido o logo do aplicativo e, logo abaixo, os botões para realizar login ou cadastro. A Figura 5.1 apresenta esse fluxo.

Figura 5.1 – Fluxo de Login/Cadastro



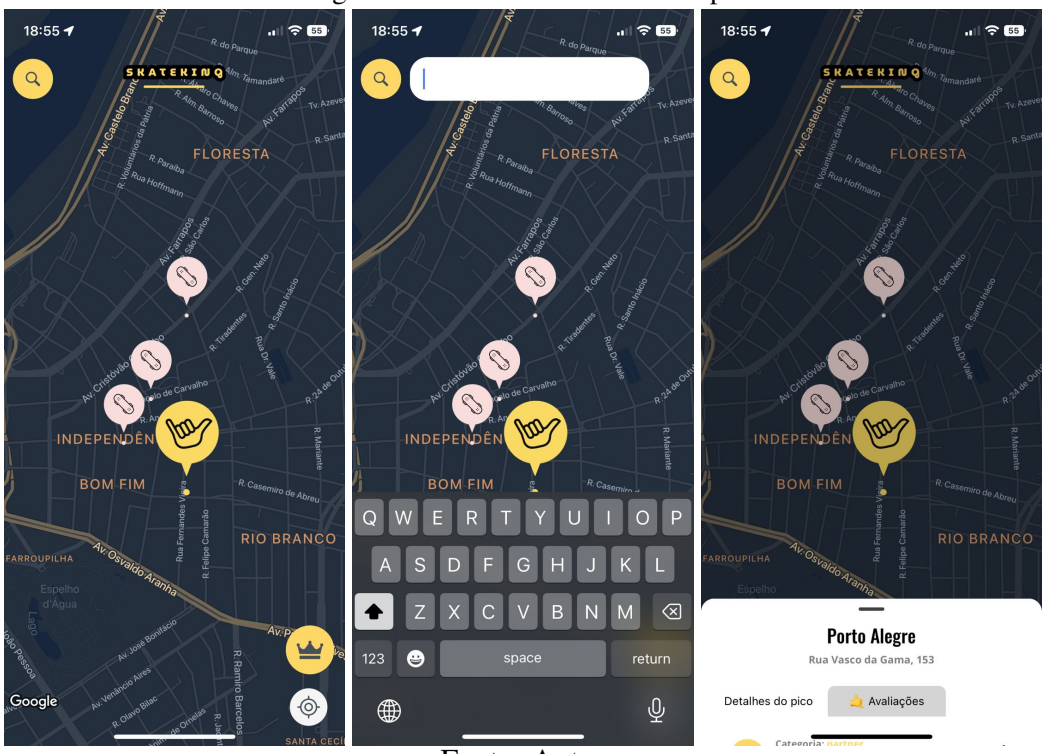
Fonte: Autor

5.1.2 Tela do Mapa

A tela do mapa é a parte principal do aplicativo. Depois de realizar o login, o usuário é direcionado para essa tela. Essa página consiste basicamente na interface do mapa centrado na posição do usuário, no qual é possível navegar realizando gestos de *drag* na tela do dispositivo. São renderizados ícones nas posições onde *picos* foram cadastrados. A cor e aparência dos ícones variam de acordo com a categoria do *pico*. As categorias possíveis são *street*, *downhill*, parceiro e *skateshop*. Ao clicar em um *pico*, é exibido um *bottom-sheet* que pode ser expandido para visualizar a tela de detalhes do *pico* (Seção 5.1.3).

Além disso, também são exibidos alguns outros elementos visuais na tela. No topo é exibido o logo do Skate King em formato de texto. No canto superior esquerdo há um botão que pode ser pressionado para realizar uma pesquisa por um endereço e, dessa forma, centrar o mapa nessa posição. No canto inferior direito são renderizados 2 botões, um deles re-centraliza o mapa na localização atual do usuário e o outro direciona para a tela de perfil (5.1.4). A Figura 5.2 mostra todos os fluxos relacionados à essa tela.

Figura 5.2 – Fluxos da Tela do Mapa



Fonte: Autor

5.1.3 Tela de Detalhes do *Pico*

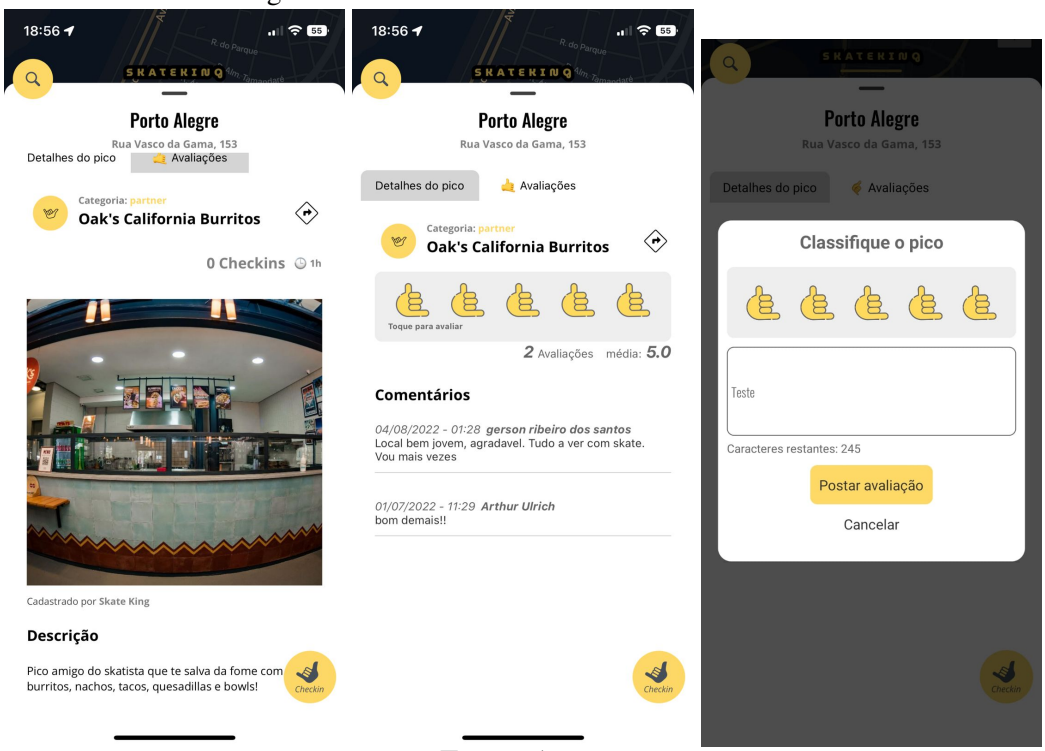
Essa tela é exibida quando o usuário expande o *bottom-sheet* depois de clicar no *pin* de um *pico*. A tela é dividida entre duas abas: detalhes do *pico* e avaliações. Na aba de detalhes, são apresentadas informações detalhadas sobre o *pico* em questão. São mostradas informações como nome do *pico*, cidade, endereço, categoria, descrição e número de *check-ins* recentes. Também é renderizada uma lista de *tags* que representa os obstáculos presentes no pico. Além disso, há um botão ao lado do nome do pico que redireciona o usuário para aplicativos de mapa com a localização do *pico* como destino. Quando o usuário é o criados do *pico* também é exibido um botão para removê-lo.

Na aba de avaliações, são apresentados o número total de avaliações, assim como a nota média. O usuário pode tocar no símbolo de avaliação para registrar uma nota de 1 a 5 e, opcionalmente, um comentário de até 250 caracteres. Esse preenchimento é realizado por um *modal* que é aberto sobre a tela. Abaixo dessas informações são listados os comentários realizados por outros usuários.

No canto inferior direito da tela, independente da aba que esteja selecionada, há um botão dedicado à realização do *check-in* no *pico*. Ao pressionado, o botão exibe uma

animação de carregamento e ao fim, caso o usuário esteja próximo ao *pico*, indica que o *check-in* foi realizado com sucesso. A Figura 5.3 apresenta a experiência do usuário em todas essas funcionalidades.

Figura 5.3 – Fluxos da Tela de Detalhes do *Pico*

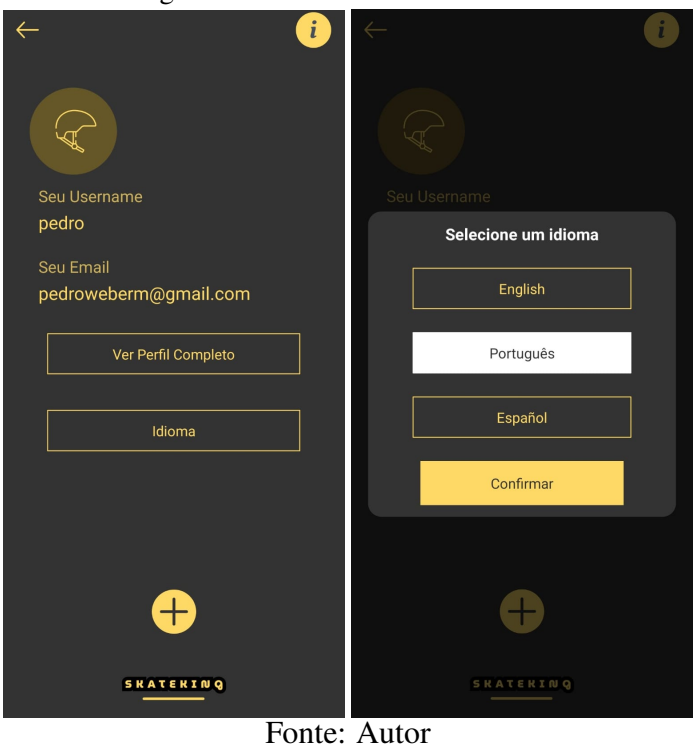


Fonte: Autor

5.1.4 Tela de Perfil

A tela de perfil exibe informações resumidas sobre o perfil cadastrado pelo usuário. No centro da tela estão posicionados dois botões, um deles leva para a tela de edição do perfil (Seção 5.1.6) e o outro possibilita que o usuário altere o idiota do aplicativo entre português, inglês e espanhol. Na parte inferior da tela há um botão que leva para a tela de criação de *pico* (Seção 5.1.5). Além disso, no canto superior direito é exibido outro botão que leva para a tela de informações (Seção 5.1.7). A Figura 5.4 mostra as interações com essa tela.

Figura 5.4 – Fluxos da Tela de Perfil

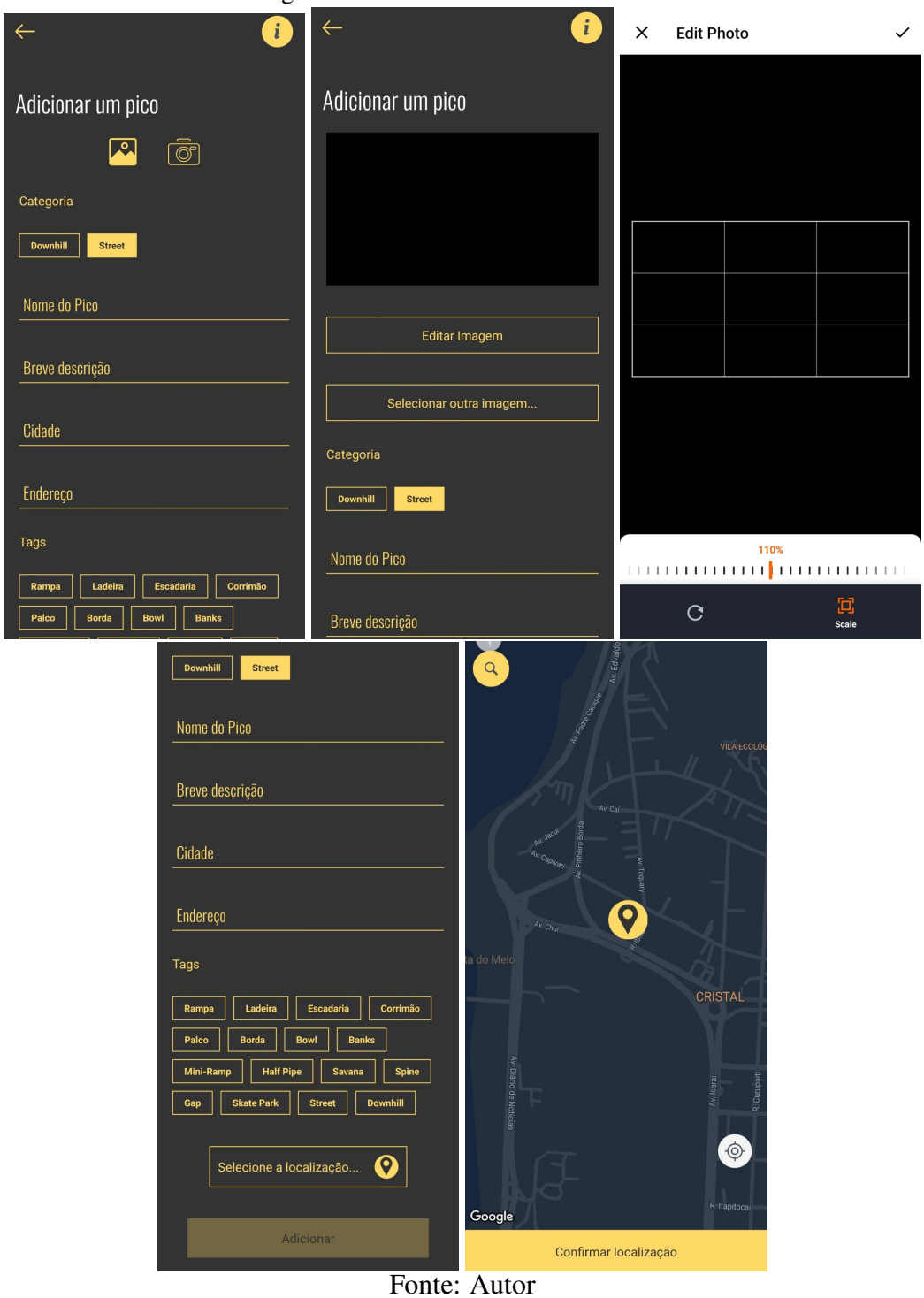


Fonte: Autor

5.1.5 Tela de Cadastro de *Pico*

Nessa tela, o usuário pode cadastrar um novo *pico* que será exibido no mapa. São dispostos na tela diversos campos para preenchimento. No topo da interface, o usuário recebe as opções de adicionar uma foto da sua galeria ou diretamente da câmera. Após adicionar a foto é possível cortar a imagem para melhorar o enquadramento. Em seguida, são solicitadas as informações de categoria, nome do *pico*, descrição, cidade, *tags* e, por fim, o usuário deve selecionar a localização do *pico* no mapa. Ao fim do cadastro, o usuário recebe um aviso sobre o sucesso ou não da criação do *pico*. Essas interfaces são demonstradas na Figura 5.5.

Figura 5.5 – Fluxo de Cadastro de *Pico*



Fonte: Autor

5.1.6 Tela de Edição de Perfil

Essa tela apresenta as informações completas sobre o perfil cadastrado do usuário. Os campos com as informações podem ser editados para atualizar o perfil, e na parte inferior é exibido um botão para salvar as alterações. Depois de salvar o usuário recebe o

feedback sobre o sucesso da operação por meio de um modal. A Figura 5.6 mostra essa tela.

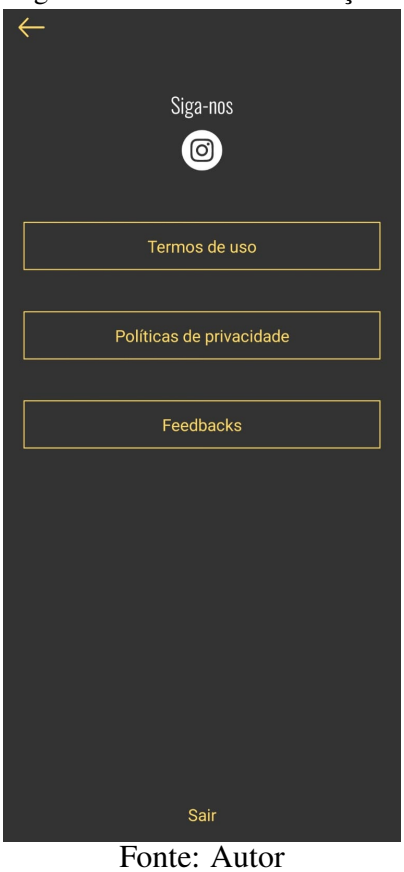
Figura 5.6 – Tela de Edição de Perfil

Fonte: Autor

5.1.7 Tela de Informações

Essa tela exibe apenas algumas informações básicas sobre o aplicativo. No topo há um botão para direcionar à página do *Instagram* do Skate King. No centro, são exibidos três botões. O primeiro e o segundo direcionam para os documentos de termos de uso e políticas de privacidade do aplicativo, respectivamente. O terceiro leva a um formulário destinado a *feedbacks* gerais sobre o aplicativo. Na parte inferior da tela está o botão destinado à ação de *logout*. Essa tela é mostrada na Figura 5.7.

Figura 5.7 – Tela de informações



Fonte: Autor

5.2 Funcionamento Módulo Gerencial

Nas seções a seguir serão apresentadas as páginas da interface web criada para servir como módulo gerencial. Esse sistema é direcionado aos mantenedores da plataforma e não aos usuários finais.

5.2.1 Página de Login

Nessa página é possível realizar o login no sistema por meio de um nome de usuário e senha. Essas credenciais são únicas e só são compartilhadas com os mantenedores da plataforma. Dessa forma, não há um fluxo destinado ao cadastro de novos usuários. Uma vez que o botão de *login* é pressionado, é exibido uma animação de carregamento e, caso as credenciais estejam corretas, ocorre um redirecionamento para a página do mapa (Seção 5.2.2). A Figura 5.8 mostra a interface dessa página.

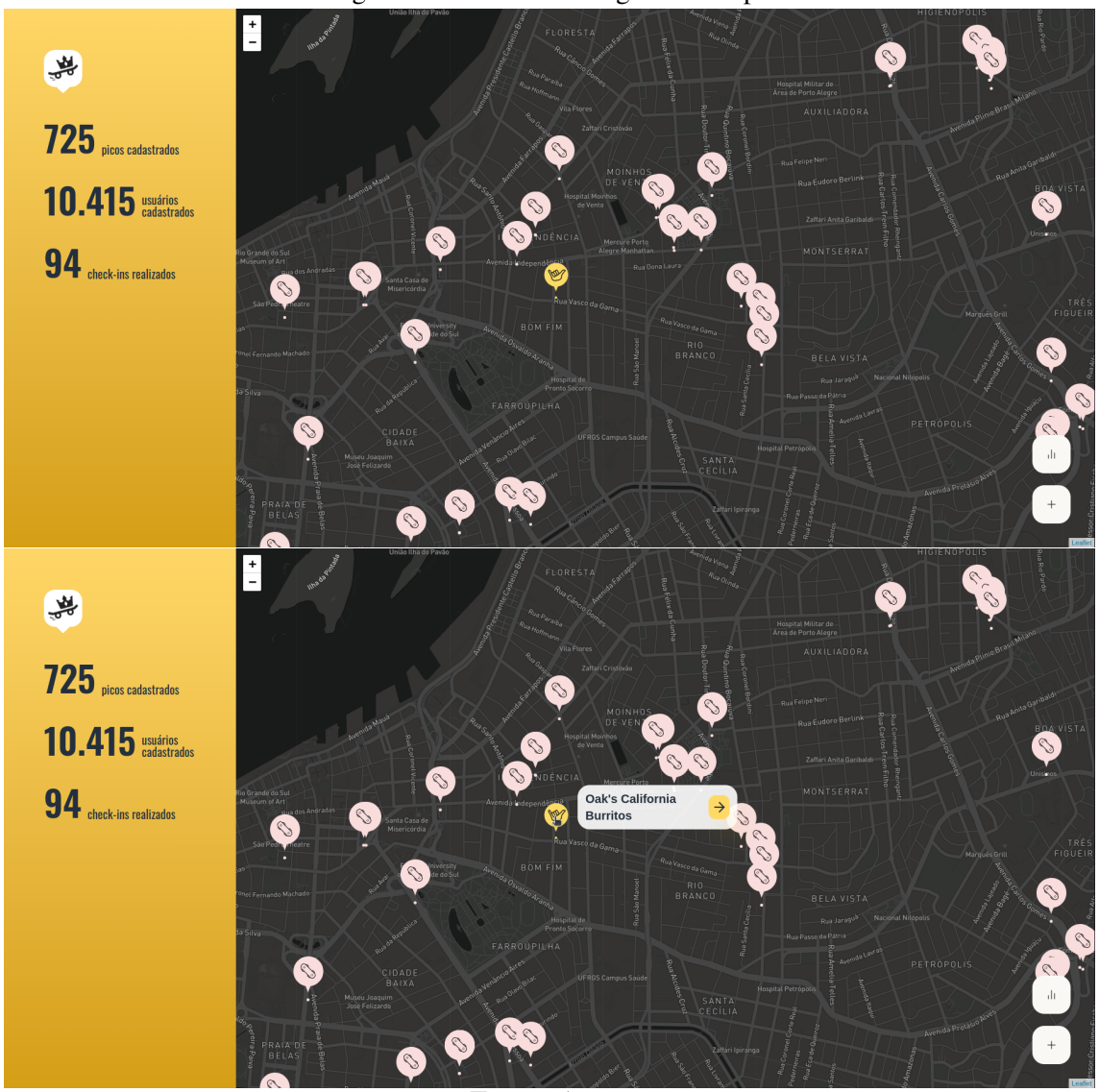
Figura 5.8 – Página de Login

Fonte: Autor

5.2.2 Página do Mapa

Essa é a página principal do módulo gerencial. Da mesma forma que no aplicativo, é exibido o mapa com ícones na posição de cada *pico*. Na lateral esquerda, são exibidas as métricas sobre o aplicativo: número de *picos* e usuários cadastrados, assim como o número de *check-ins* realizados. Ao clicar no ícone de um *pico* é exibido o nome do *pico* e um botão para ir à tela de detalhe (Seção 5.2.4) . No canto inferior direito são exibidos dois botões. Um deles é destinado à uma página de métricas detalhadas que, no momento da escrita desse trabalho, ainda não foi implementada. Dessa forma, o botão não executa nenhuma ação ao ser pressionado. O segundo botão direciona para a página de criação de um *pico* (Seção 5.2.3). A Figura 5.9 mostra a interface dessa página.

Figura 5.9 – Fluxos da Página do Mapa

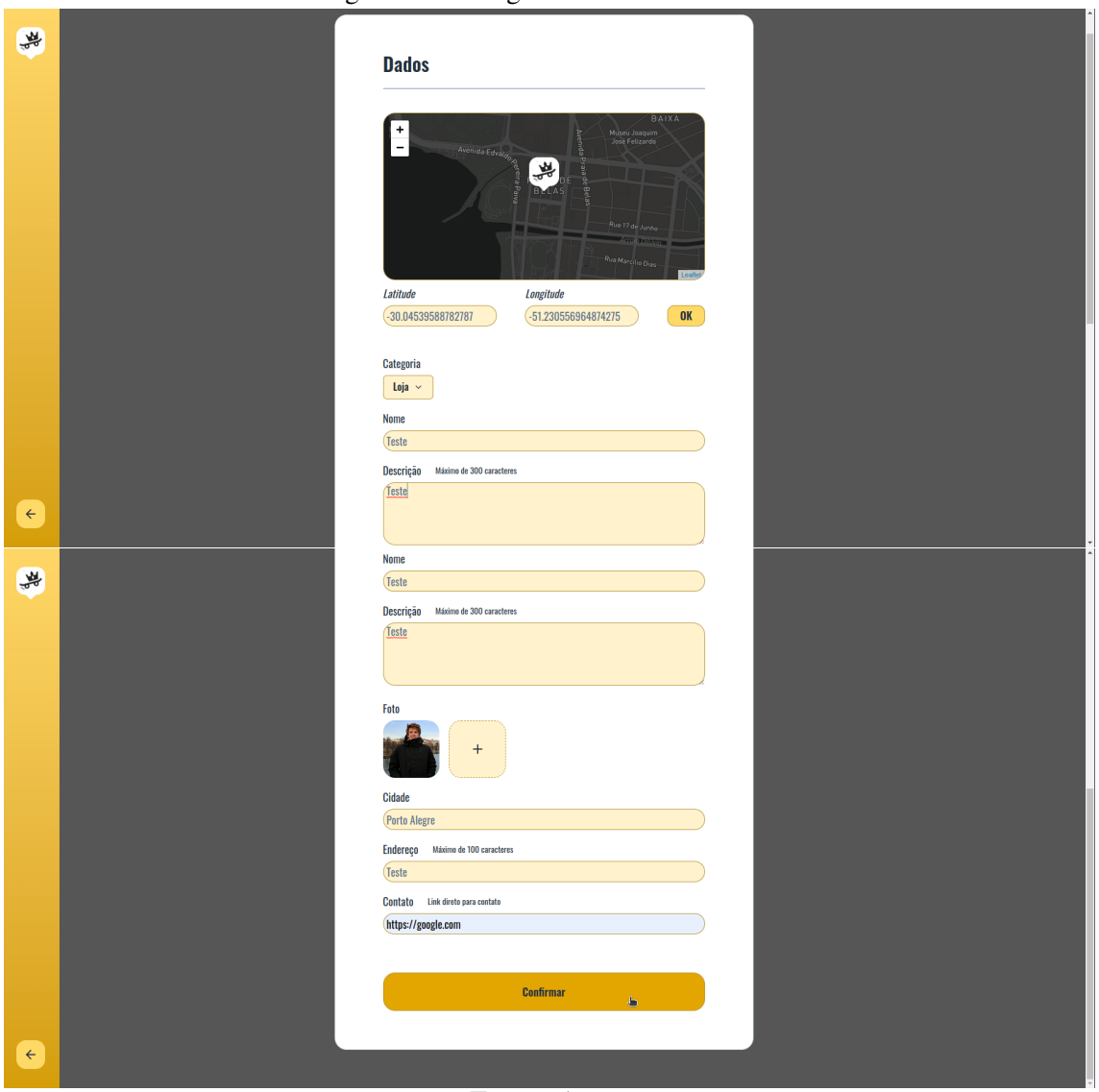


Fonte: Autor

5.2.3 Página de Cadastro de *Pico*

Essa página funciona de forma bastante semelhante a tela de cadastro de *pico* do aplicativo. São exibidos os campos para preenchimento dos dados sobre o *pico*. No topo da página é exibida uma miniatura do mapa onde é possível marcar o local com um *pin* ou, opcionalmente, inserir os valores de latitude e longitude diretamente. Em seguida, estão os campos destinados à categoria, nome, descrição, foto, cidade, endereço e, caso a categoria seja *Loja*, um campo para um *link* de contato da loja. Ao pressionar o botão de confirmação, é exibida uma caixa de diálogo com o *feedback* sobre o resultado da operação. Na Figura 5.10 é apresentado o funcionamento dessa página.

Figura 5.10 – Página de Cadasto de *Pico*

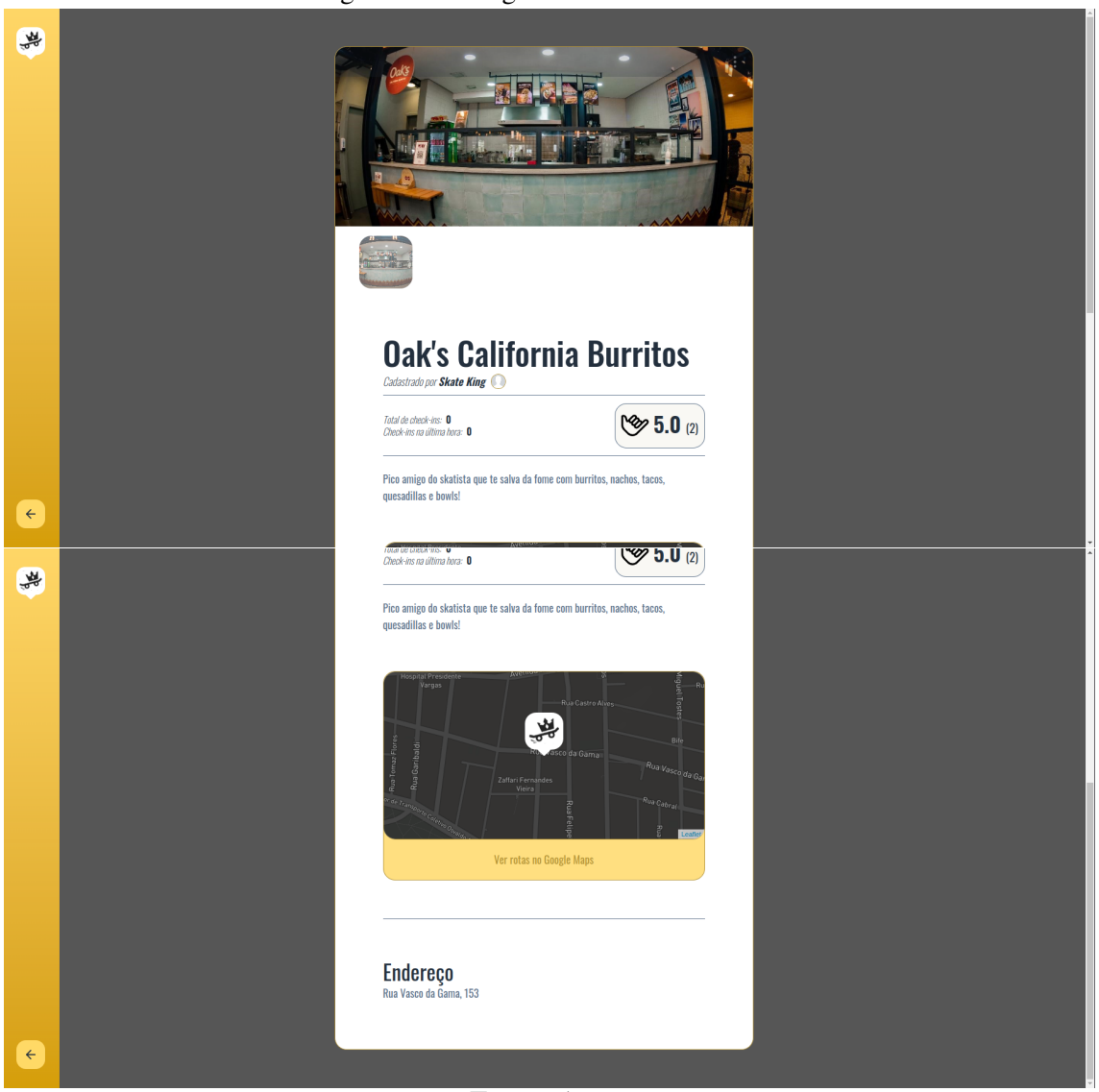


Fonte: Autor

5.2.4 Página de Detalhes do *Pico*

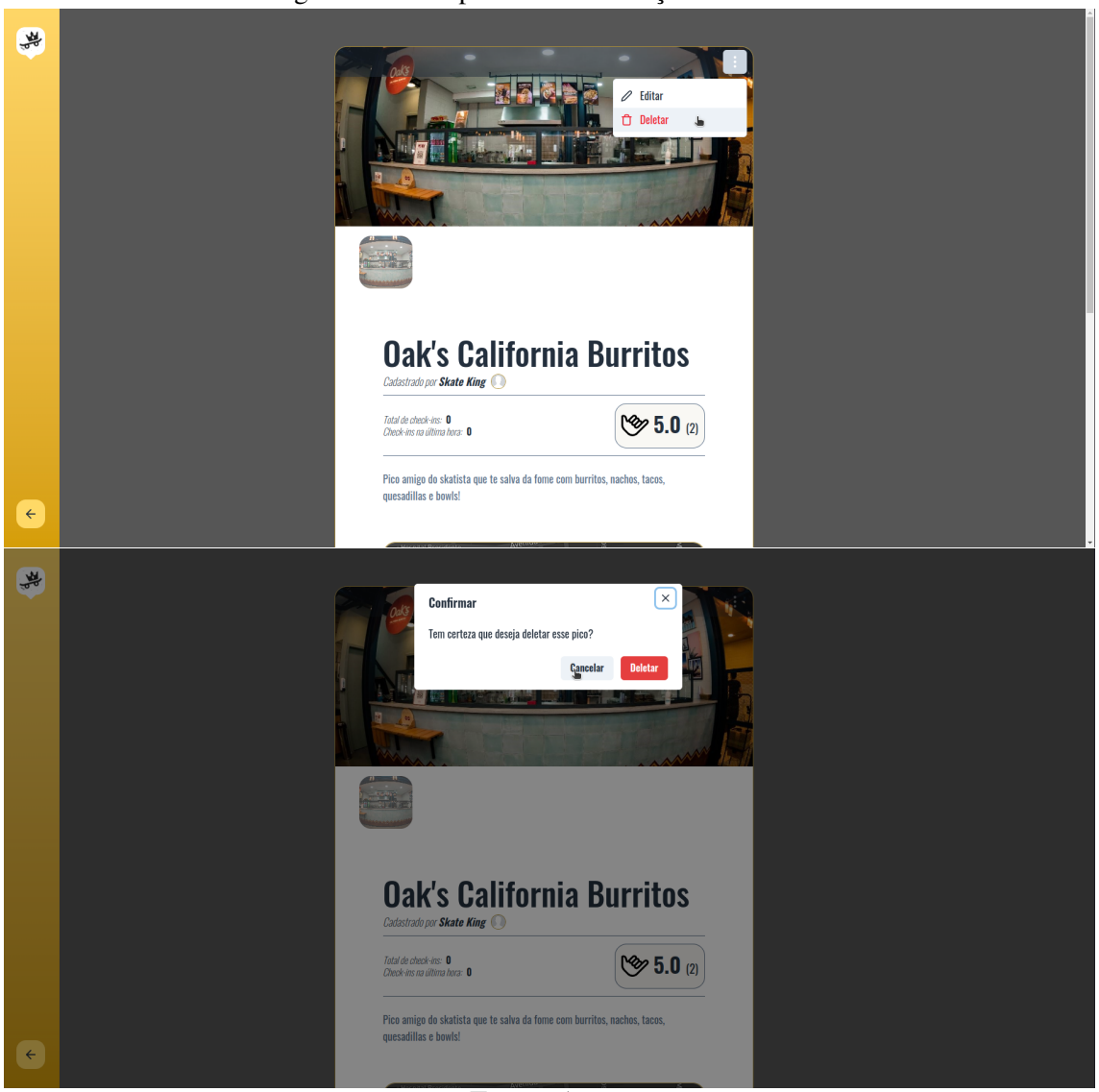
Nessa página, são apresentadas informações detalhas sobre um pico. Primeiramente é exibida a foto, seguida do nome do *pico*, nome do criador, número de *check-ins* recentes e totais, nota média e total de avaliações, descrição, local no mapa e endereço. No canto superior direito há um botão que, ao clicado, exibe as opções de editar ou deletar o *pico*. A opção de editar não realiza nenhuma ação, já que, no momento da escrita desse trabalho, essa operação não foi implementada. Ao clicar no botão de deleção, é exibido um modal de confirmação e, se a operação for confirmada, é exibida uma caixa de diálogo com o resultado da requisição. As Figuras 5.11 e 5.12 mostram essa experiência.

Figura 5.11 – Página de Detalhes do *Pico*



Fonte: Autor

Figura 5.12 – Experiência de Deleção de um *Pico*



Fonte: Autor

6 AVALIAÇÃO COM USUÁRIOS

O objetivo desse capítulo é apresentar o experimento realizado com usuários que teve o intuito de avaliar a experiência e o desempenho do sistema. O experimento consiste em uma série de tarefas a serem realizadas dentro do aplicativo e, para cada tarefa, a avaliação pelo usuário sobre facilidade de execução. Ao final do Capítulo são apresentados os resultados do experimento assim como uma análise sobre melhorias que podem ser implementadas.

6.1 Ambiente dos Experimentos

O objetivo deste capítulo é realizar avaliações com possíveis usuários do sistema com o intuito de verificar se as premissas propostas para o aplicativo foram satisfeitas. O experimento foi aplicado por meio de um formulário *online*. Os usuários devem abrir o formulário, em seu dispositivo móvel ou computador, juntamente com o aplicativo, e seguir as instruções descritas. O formulário foi enviado em grupos de *Whatsapp* dedicados à comunidade de usuários do aplicativo, assim como diretamente para alguns usuários específicos.

6.2 Protocolo de Testes

Para cada usuário, foram apresentadas as etapas a seguir:

- **Formulário pré-teste**: Nessa etapa, foram realizadas perguntas com o intuito de caracterizar os usuários. As perguntas foram: "Qual sua idade?", "Você tem o hábito de andar de skate?" e "Qual seu nível de familiaridade com *smartphones* e aplicativos móveis?".
- **Treinamento**: Nessa etapa, o funcionamento e objetivo do aplicativo foram descritos. Ao fim da etapa, foi solicitada uma tarefa simples que não foi avaliada. Após cada item foi solicitada uma confirmação de que o usuário leu a informação. Os passos apresentados nessa etapa foram:

    1. O Skate King é um aplicativo que possibilita que praticantes do skate encontrem e cadastrem locais que são propícios para a prática do esporte. Esses

locais são chamados de "picos".

2. No Skate King, você pode navegar pelo mapa, visualizando os picos que foram cadastrados por outros skatistas assim como cadastrar seus próprios picos. Além disso, é possível fazer check-in e avaliar um pico com uma nota de 1 a 5.
3. Para realizar esse teste você precisará fazer o download do aplicativo no seu celular iOS ou Android. Você pode fazer isso por aqui ou pela sua loja de aplicativos: Android- *Link para download*, iOS- *Link para download*.
4. Quando você abrir o aplicativo pela primeira vez, será necessário fazer o cadastro usando sua conta Google ou Apple. É bem rápido! Caso você já tenha se cadastrado em outro momento, terá apenas que fazer o login.
5. Antes de começar, vamos fazer um "aquecimento"para que você se familiarize com o aplicativo.
6. A partir da tela principal (mapa) navegue pela região até encontrar algum pico. Os picos são representados por ícones coloridos que ficam em destaque no mapa. Quando encontrar, clique no ícone para visualizar informações sobre esse pico. Você conseguiu realizar essa tarefa?
7. Qual o nome do pico que você encontrou? Caso não tenha encontrado você pode pular essa pergunta.
8. Agora que você já conhece o aplicativo, vamos iniciar a avaliação. Serão descritas várias tarefas para que você realize. Depois de cada tarefa, você avaliará com uma nota de 1 a 5 o quão fácil foi realizar a tarefa e poderá deixar algum comentário se quiser.

- **Avaliação**: Para essa etapa, foram definidas diversas tarefas a serem realizadas no aplicativo. Para cada tarefa, é descrito o que deve ser feito e, a seguir, são feitas as seguintes perguntas: "Você conseguiu realizar essa tarefa?", "Quão fácil foi realizar essa tarefa (1 a 5)?", "Se quiser, deixe um comentário sobre a experiência de realizar essa tarefa.". As descrições das tarefas solicitadas foram as seguintes:

  1. Crie um novo pico na sua localização atual. Adicione um nome, uma foto e uma descrição nesse pico. É importante que o pico seja criado bem próximo de onde você está realizando essa avaliação. Não se preocupe com o que irá escrever, pois você removerá esse pico ao fim da avaliação.
  2. Encontre o pico que você acabou de cadastrar e visualize as informações dele.

Veja o nome, foto e descrição do pico.

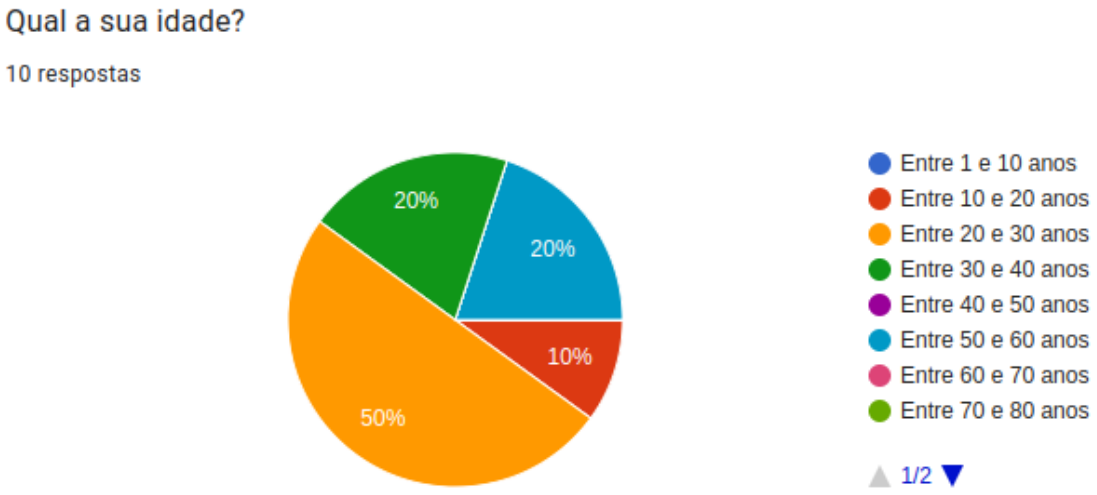
3. Encontre novamente o pico que você cadastrou e realize um check-in nele.
4. Encontre novamente o pico que você cadastrou e visualize quantos check-ins foram realizados nele recentemente.
5. Encontre novamente o pico que você cadastrou e faça uma avaliação dele. Escolha uma nota de 1 a 5 e escreva um comentário.
6. Encontre novamente o pico que você cadastrou e visualize as avaliações dele. Se você conseguiu realizar a avaliação no passo anterior, você deve conseguir ver uma nota média assim como o seu comentário.
7. Encontre novamente o pico que você cadastrou e, dessa vez, exclua esse pico. Você só conseguirá fazer isso se o pico foi criado por você.

- **Comentários**: Na última etapa foi disponibilizado um campo livre para comentários gerais sobre o teste ou sobre o aplicativo.

6.3 Perfil dos Usuários

Com as perguntas realizadas na primeira etapa do teste, foi possível identificar a demografia do grupo que realizou o teste. Como pode ser visto na Figura 6.1, a grande maioria dos avaliadores tem entre 20 e 30 anos, com alguns usuários nas faixas de 30 a 40 e 50 a 60 e apenas um na faixa de 10 a 20. Na Figura 6.2, podemos ver a proporção de usuários que costumam praticar o skate. Por fim, na Figura 6.3 vemos que a maioria dos avaliadores tem um nível alto de familiaridade com *smartphones* e aplicativos móveis.

Figura 6.1 – Faixa etária dos usuários

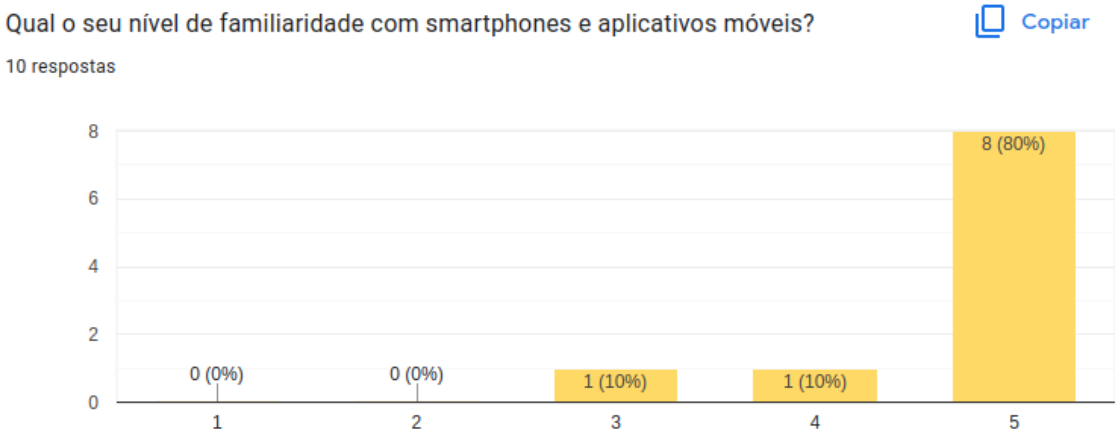


Fonte: Autor

Figura 6.2 – Proporção de usuários que praticam o skate

Fonte: Autor

Figura 6.3 – Familiaridade dos usuários com *smartphones* e aplicativos móveis



Fonte: Autor

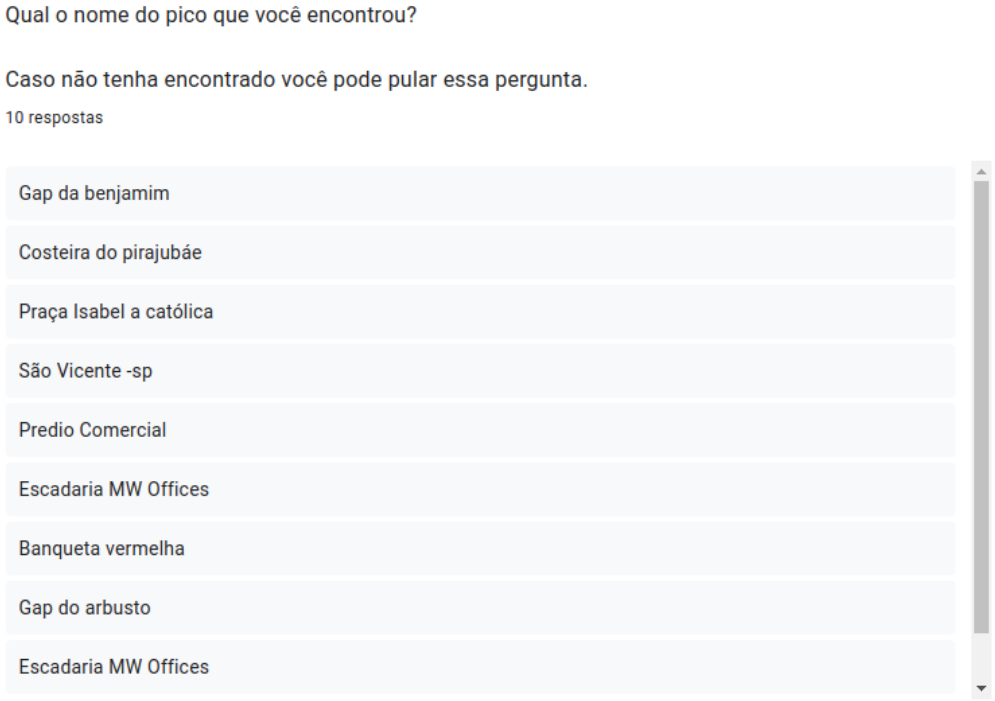
6.4 Análise dos Resultados

Nessa Seção, serão apresentados, para cada etapa do teste, os resultados obtidos. Também será feito um apanhado geral sobre os comentários presentes nas respostas de perguntas de campo livre. Por fim, com base nos resultados, serão apresentados os principais pontos de melhoria para o sistema.

6.4.1 Treinamento

Todos os usuários marcaram os itens confirmando a leitura das informações sobre o aplicativo. Da mesma forma, todos conseguiram realizar o cadastro ou *login* e acessar o aplicativo. Na tarefa solicitada, todos os usuários encontraram algum *pico* na sua região e responderam à ultima pergunta com o nome do *pico* encontrado (Figura 6.4).

Figura 6.4 – Picos encontrados pelos usuários

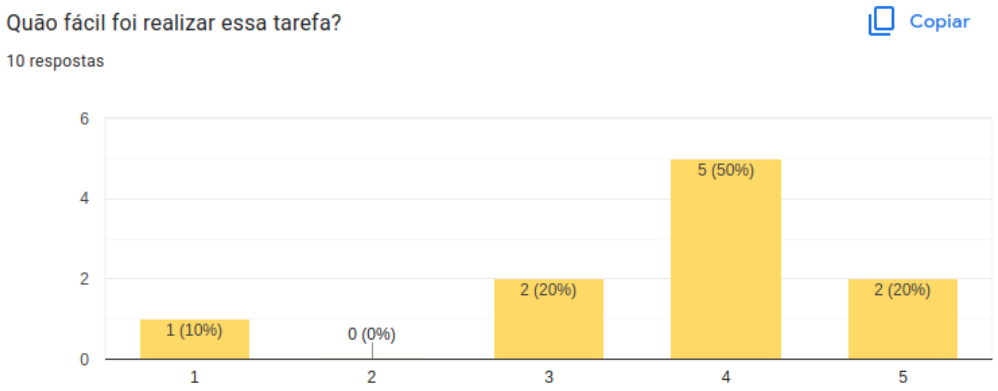


Fonte: Autor

6.4.2 Tarefa 1

- **Descrição da tarefa:** Crie um novo pico na sua localização atual. Adicione um nome, uma foto e uma descrição nesse pico. É importante que o pico seja criado bem próximo de onde você está realizando essa avaliação.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?
  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?
  Resultado (Figura 6.5): 1 (10%), 2 (0%), 3 (20%), 4 (50%), 5 (20%)

Figura 6.5 – Resultados Tarefa 1



Fonte: Autor

- **Pergunta 3**: Comentários

  Resultado: Para essa tarefa, os comentários mencionaram, principalmente, sobre o fato de que o botão da coroa leva a um menu onde o *pico* pode ser cadastrado não é intuitivo. Além disso, também foi comentado sobre a falta de clareza sobre quais campos são obrigatórios no preenchimento dos dados do *pico*.

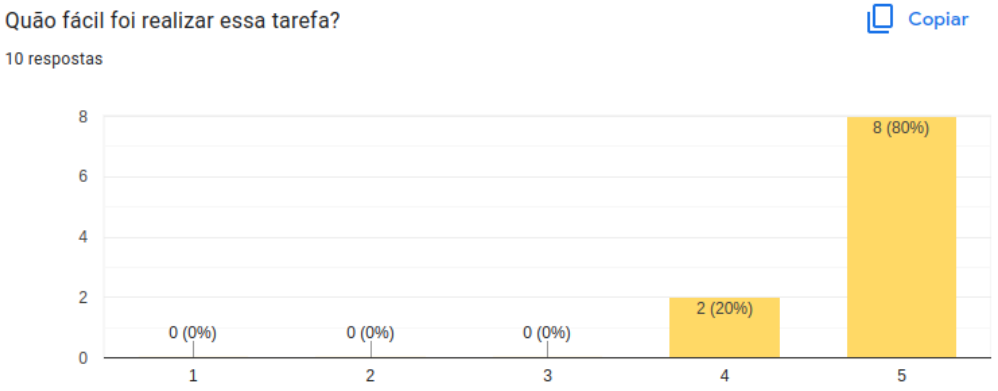
6.4.3 Tarefa 2

- **Descrição da tarefa:** Encontre o pico que você acabou de cadastrar e visualize as informações dele. Veja o nome, foto e descrição do pico.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?

  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?

  Resultado (Figura 6.6): 1 (0%), 2 (0%), 3 (0%), 4 (20%), 5 (80%)

Figura 6.6 – Resultados Tarefa 2



Fonte: Autor

- **Pergunta 3**: Comentários

  Resultado: Não houve comentários relevantes.

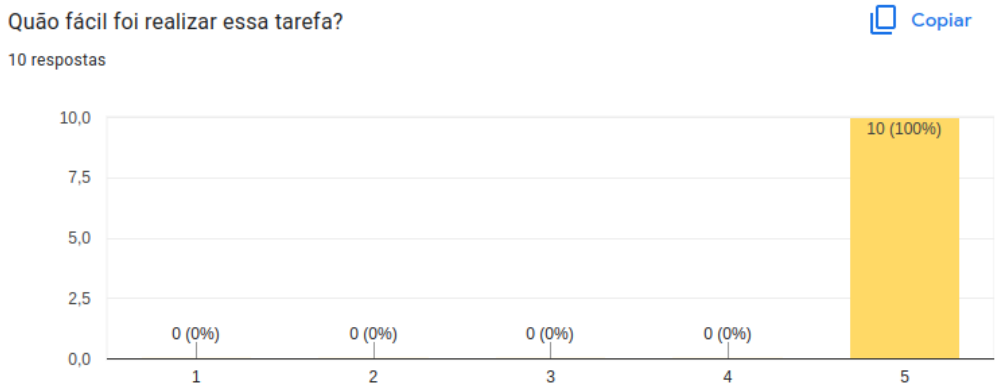
6.4.4 Tarefa 3

- **Descrição da tarefa:** Encontre novamente o pico que você cadastrou e realize um check-in nele.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?

  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?

  Resultado (Figura 6.7): 1 (0%), 2 (0%), 3 (0%), 4 (0%), 5 (100%)

Figura 6.7 – Resultados Tarefa 3



Fonte: Autor

- **Pergunta 3**: Comentários

  Resultado: Houve apenas um comentário sugerindo que o botão de *check-in* fosse maior.

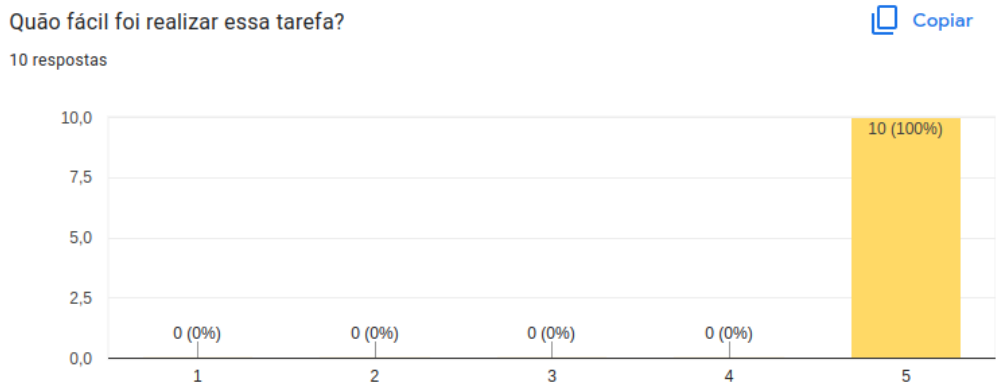
6.4.5 Tarefa 4

- **Descrição da tarefa:** Encontre novamente o pico que você cadastrou e visualize quantos check-ins foram realizados nele recentemente.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?

  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?

  Resultado (Figura 6.8): 1 (0%), 2 (0%), 3 (0%), 4 (0%), 5 (100%)

Figura 6.8 – Resultados Tarefa 4



Fonte: Autor

- **Pergunta 3**: Comentários

  Resultado: Não houve comentários relevantes.

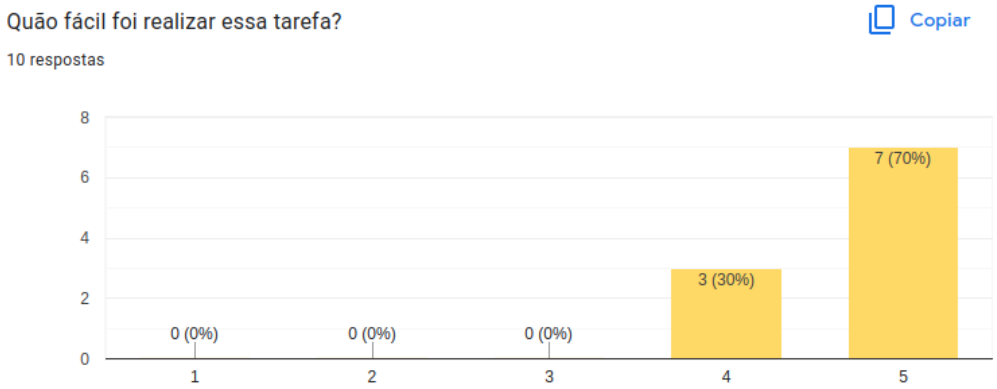
6.4.6 Tarefa 5

- **Descrição da tarefa:** Encontre novamente o pico que você cadastrou e faça uma avaliação dele. Escolha uma nota de 1 a 5 e escreva um comentário.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?

  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?

  Resultado (Figura 6.9): 1 (0%), 2 (0%), 3 (0%), 4 (30%), 5 (70%)

Figura 6.9 – Resultados Tarefa 5



Fonte: Autor

- **Pergunta 3**: Comentários

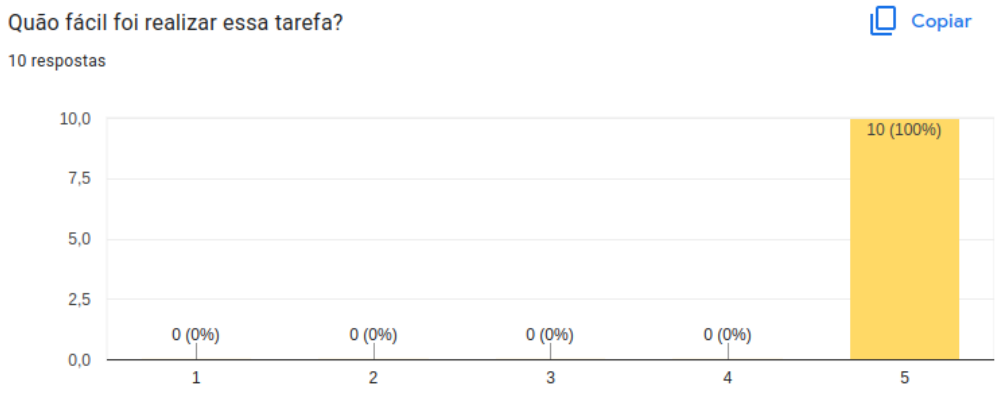
  Resultado: Foi comentado sobre a necessidade de clicar 2 vezes no ícone de ava-

liação, sugerindo que a nota fosse automaticamente preenchida depois do primeiro clique.

6.4.7 Tarefa 6

- **Descrição da tarefa:** Encontre novamente o pico que você cadastrou e visualize as avaliações dele. Se você conseguiu realizar a avaliação no passo anterior, você deve conseguir ver uma nota média assim como o seu comentário.
- **Pergunta 1**: Você conseguiu realizar essa tarefa?
  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?
  Resultado (Figura 6.10): 1 (0%), 2 (0%), 3 (0%), 4 (0%), 5 (100%)

Figura 6.10 – Resultados Tarefa 6



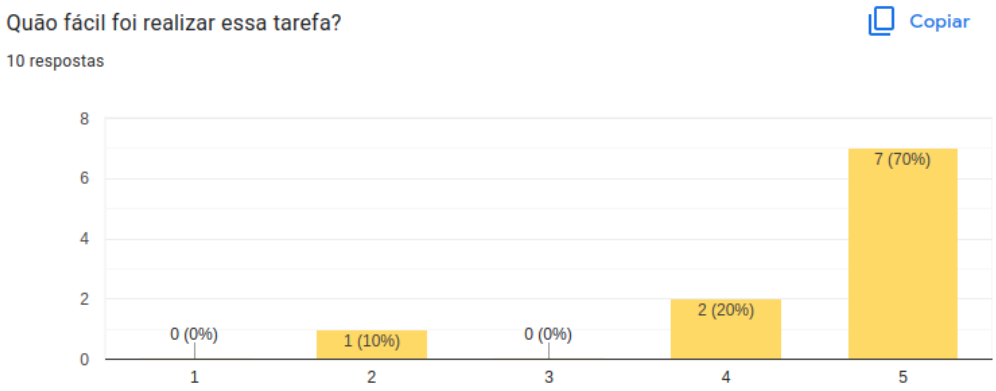
Fonte: Autor

- **Pergunta 3**: Comentários
  Resultado: Não houve comentários relevantes.

6.4.8 Tarefa 7

- **Descrição da tarefa:** Encontre novamente o pico que você cadastrou e, dessa vez, exclua esse pico. Você só conseguirá fazer isso se o pico foi criado por você
- **Pergunta 1**: Você conseguiu realizar essa tarefa?
  Resultado: Sim (100%), Não (0%)
- **Pergunta 2**: Quão fácil foi realizar essa tarefa?
  Resultado (Figura 6.11): 1 (0%), 2 (10%), 3 (0%), 4 (20%), 5 (70%)

Figura 6.11 – Resultados Tarefa 7



Fonte: Autor

- **Pergunta 3**: Comentários

  Resultado: Foi sugerido que o botão para remover um pico fosse maior e mais visível. Além disso, também foi relatado que há uma inconsistência na tela que é exibida após a remoção do *pico*.

6.4.9 Análise

Como pode ser visto nos resultados apresentados, todos os usuários conseguiram realizar as tarefas propostas e as avaliações tiveram notas, em sua maioria, altas. Foram identificados pontos de melhoria para versões futuras do aplicativo. Algumas evoluções que pode ser citadas são a mudança do fluxo para chegar até a tela de cadastro de *pico*, melhoria da clareza sobre os campos obrigatórios no cadastro de *pico*, aumento do tamanho do botão de *check-in*, melhoria do fluxo para criação de uma avaliação e o aumento do tamanho do botão destinado à remoção de um *pico*.

7 CONCLUSÃO

Este trabalho apresentou o desenvolvimento e a implantação da plataforma Skate King, um sistema dedicado ao mapeamento e descoberta de locais propícios para a prática do skate, chamados de *picos*. Com o aplicativo Skate King, os skatistas podem cadastrar seus *picos* favoritos com nome, foto, descrição e obstáculos. Além disso, é possível descobrir novos *picos* cadastrados por outros *skatistas* e conhecer novos lugares. Os usuários podem também realizar *check-in* em *pico* e cadastrar dados no seu perfil de *skatistas*. Os *picos* podem também ser avaliados com uma nota de 1 a 5 e um comentário.

No texto foram documentados os principais padrões, tecnologias e metodologias de desenvolvimento de *software* utilizados. Foram definidos pilares que guiaram a implementação do sistema e as decisões técnicas. O trabalho apresentou os detalhes do desenvolvimento de duas aplicações *front-end* e uma aplicação *back-end* implantada na nuvem AWS. Além disso, foram apresentados os desafios da implementação de sistemas que lidam com dados de geolocalização e a forma como foram solucionados.

Após a implementação, foi realizado um experimento com usuários. Com esse experimento, foi possível avaliar a qualidade da experiência do usuário ao utilizar o aplicativo, além do desempenho e consistência do sistema como um todo. Com o resultado do experimento, identificou-se que a experiência geral de uso do aplicativo implementado foi satisfatória. No entanto, foram relatados pontos de possíveis melhorias para uma experiência mais fluida que podem ser implementadas em versões futuras.

Com esse trabalho, é definida uma referência que pode ser usada como base por outros desenvolvedores que tenham o objetivo de implementar sistemas escaláveis de forma rápida e eficiente. Foram definidos processos importantes para a criação de aplicações implantadas em nuvem. Além disso, foram documentados padrões para a criação de aplicações que utilizam dados de geolocalização. Ainda, o aplicativo desenvolvido foi disponibilizado de forma gratuita nas lojas de aplicativos e pode ser utilizado livremente por skatistas que tenham interesse.

O sistema implementado continuará sendo evoluído em suas versões futuras. Com a avaliação com usuários, foram identificadas melhorias importantes que devem ser priorizadas para atualizações futuras. Algumas dessas melhorias são a mudança do fluxo para chegar até a tela de cadastro de *pico*, melhoria da clareza sobre os campos obrigatórios no cadastro de *pico*, aumento do tamanho do botão de *check-in*, melhoria do fluxo para criação de uma avaliação e o aumento do tamanho do botão destinado à remoção de um

pico. Além disso, novas funcionalidades já estão sendo planejadas e devem ser adicionadas em novas versões. Algumas delas são: sistema de recompensas, criação de eventos e gerenciamento de grupos.

REFERÊNCIAS

ALEXANDER, M. **SK8 Spots**. mike_zander, 2021. Disponível em: <https://play.google.com/store/apps/details?id=michaelalexander.SkateSpots>.

ARTIUM. **Smap - Skateparks, skate spots**. Artium, 2022. Disponível em: <https://play.google.com/store/apps/details?id=org.nativescript.rideNRoll>.

ATTABOY. **Loke: Skate spots challenges**. Attaboy makes, 2022. Disponível em: <https://play.google.com/store/apps/details?id=com.loke>.

AWS. **Infraestrutura global da AWS**. Amazon Web Services, 2022. Disponível em: <https://aws.amazon.com/pt/about-aws/global-infrastructure/>.

AWS. **Modelo de responsabilidade compartilhada**. Amazon Web Services, 2022. Disponível em: <https://aws.amazon.com/pt/compliance/shared-responsibility-model/>.

DAHL, R. OpenJS Foundation, 2009. Disponível em: <https://nodejs.org/>.

ECMA International. **Standard ECMA-262 - ECMAScript Language Specification**. 13. ed. [s.n.], 2022. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-262/>.

ESTES, B. **Geolocation-the risk and benefits of a trending technology**. ISACA, 2016. Disponível em: <https://www.isaca.org/resources/isaca-journal/issues/2016/volume-5/geolocationthe-risk-and-benefits-of-a-trending-technology>.

EVANS, E.; FOWLER, M. **Domain-driven design tackling complexity in the heart of software**. [S.l.]: Addison-Wesley, 2003.

FIELDING, R. T. **Architectural Styles and the Design of Network-Based Software Architectures**. Tese (Doutorado), 2000. AAI9980887.

FOWLER, M. **DomainDrivenDesign**. 2020. Disponível em: <https://martinfowler.com>.

GOOGLE. **Google Maps**. Google, 2022. Disponível em: <https://play.google.com/store/apps/details?id=com.google.android.apps.maps>.

HILL, M. D. What is scalability? **Scalable Shared Memory Multiprocessors**, p. 89–96, 1992.

INTERNATIONAL, E. **The JSON data interchange syntax**. ECMA International, 2017. Disponível em: <https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf>.

IOC. **Skateboarding**. International Olympic Committee, 2020. Disponível em: <https://olympics.com/en/sports/skateboarding/>.

MACHADO, G. M. C. **De "carrinho"pela cidade: a prática do street skate em São Paulo**. Dissertação (Mestrado) — Faculdade de Filosofia, Letras e Ciências Humanas, 2011. Disponível em: <https://doi.org/10.11606/d.8.2011.tde-05062012-160404>.

MARTIN, M. **Skateboarding history: From the backyard to the big time**. [S.l.]: Edge Books, 2005.

MARTIN, R. C. **The Clean Architecture**. 2012. Disponível em: <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>.

MARTIN, R. C.; GRENNING, J.; BROWN, S. **Clean architecture: A craftsman's guide to software structure and Design**. [S.l.]: Prentice Hall, 2018.

MICROSOFT. **JavaScript with syntax for types.** 2012. Disponível em: <https://www.typescriptlang.org/>.

NIANTIC. **Pokémon GO**. Niantic, Inc., 2022. Disponível em: <https://play.google.com/store/apps/details?id=com.nianticlabs.pokemongo>.

NOOR, A. **How javascript is quickly becoming the market leader**. 2022. Disponível em: <https://www.mobilelive.ca/blog/javascript-leader>.

PATTON, J. et al. **User story mapping: Discover the whole story, build the right product**. [S.l.]: O'Reilly, 2014.

SCHWABER, K.; SUTHERLAND, J. **The Scrum Guide**. [s.n.], 2020. Disponível em: <https://scrumguides.org/>.

SKATE, H. H. **Hubba Skate Spots**. Hubba Hubba Skate LLC, 2017. Disponível em: <https://play.google.com/store/apps/details?id=com.hubbahubba.android>.

SWAIL, P. **Serverless Glossary**. 2019. Disponível em: <https://serverlessfirst.com/serverless-glossary/>.

TERAVAINEN, T. **What is single sign-on (SSO) and how does it work?** TechTarget, 2020. Disponível em: <https://www.techtarget.com/searchsecurity/definition/single-sign-on>.

WHATWG. **HTML specification**. [S.l.], 2022. Https://html.spec.whatwg.org/.

WIGGINS, A. **The twelve-factor app**. 2017. Disponível em: <https://12factor.net/>.