

UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
CAMPUS SANTA MÔNICA  
CURSO DE ENGENHARIA DA COMPUTAÇÃO

Igor Gonçalves Ribeiro Silva

## **Aplicação Mobile Serverless**

Uberlândia, Minas Gerais

2020

Igor Gonçalves Ribeiro Silva

## **Aplicação Mobile Serverless**

Trabalho de Conclusão apresentado ao Curso de Engenharia da Computação como requisito parcial para a obtenção do título de Bacharel em Engenharia da Computação.

Universidade Federal de Uberlândia

Orientador: Marcelo Rodrigues de Sousa

Uberlândia, Minas Gerais

2020



SERVIÇO PÚBLICO FEDERAL  
MINISTÉRIO DA EDUCAÇÃO  
UNIVERSIDADE FEDERAL DE UBERLÂNDIA  
FACULDADE DE ENGENHARIA ELÉTRICA  
COORDENAÇÃO DO CURSO DE GRADUAÇÃO EM  
ENGENHARIA ELÉTRICA



ATA DE DEFESA DE TRABALHO DE CONCLUSÃO DE CURSO

Aluno (a): IGOR GONÇALVES RIBEIRO SILVA

Matrícula: 11511ECP014

Data da Defesa: 03/07/2019 Horário de início: 15H Local: SALA VERMELHA (ANEXO)

Título da Monografia:

"Aplicação Mobile Serverless."

Banca Examinadora:

Orientador: Dr. Marcelo Rodrigues de Sousa

Membro1: Dr. Igor Santos Peretta

Membro2: Dr. Márcio José da Cunha

Em sessão pública, após assistir a exposição oral de cerca de 30 minutos, e após arguir o estudante por 20 minutos, a banca deliberou pela (X) APROVAÇÃO ( ) REPROVAÇÃO do(a) aluno(a). Na forma regular, foi lavrada a presente ata, assinada pelos membros da banca e pelo(a) estudante, que será encaminhada ao Coordenador de Trabalho de Conclusão de Curso para registro e verificação do cumprimento, pelo(a) estudante, de eventuais alterações requisitadas pelos membros da banca, listadas no campo de observações abaixo. Uberlândia, MG, 03 de julho de 2019.

Observações:

Realizar as correções sugeridas pela banca.

Assinaturas:

Componentes da banca:

[Assinatura]  
[Assinatura]  
[Assinatura]

Estudante: Igor G. R. Silva

Dedico este trabalho à minha família,  
pelos momentos de ausência.

## **AGRADECIMENTOS**

Agradeço primeiramente à minha família, pois que sem tal apoio seria impossível vencer esse desafio. Peço também desculpas pelos momentos de ausência, os quais nesses últimos cinco anos foram tão frequentes e contínuos.

Agradeço ao meu orientador Prof. Dr. Marcelo Rodrigues, pela sabedoria com que me guiou nesta trajetória.

A Secretaria do Curso, pela cooperação.

Certamente estes parágrafos não atendem a todas as pessoas que fizeram parte dessa importante fase de minha vida. Portanto, desde já peço desculpas àquelas que não estão presentes entre essas palavras, mas elas podem estar certas que fazem parte do meu pensamento e de minha gratidão.

Enfim, a todos os que, por algum motivo e de alguma maneira, contribuíram para a realização deste trabalho.

*“Educação não é gasto,  
é investimento com retorno garantido.”  
(Sir Arthur Lewis)*

## RESUMO

À medida que a computação avança novas formas de se desenvolver sistemas são discutidas a fim de que a sustentabilidade esteja sempre presente e seja cada vez mais aprimorada. A computação em nuvem veio se desenvolvendo de tal forma que, atualmente, são oferecidos recursos para implantação de sistemas sustentáveis, escaláveis e acessíveis a todos os que queiram desenvolver, além de um baixo custo para o investimento inicial. Este trabalho tem por objetivo desenvolver uma aplicação *mobile* com todo o processamento de *backend* sendo feito na nuvem, mantendo a confiabilidade, baixa latência, escalabilidade o que irá garantir uma ótima experiência ao usuário. Serão apresentados e discutidos métodos de arquitetura sem servidor e controle de acesso na nuvem, além de tecnologias como bancos de dados não relacionais e execução de código sem provisionamento de servidores. O sistema proposto será implantado na nuvem da Amazon Web Service.

**Palavras-chave:** Computação em nuvem. Serverless. IaaS. Microserviços

## ABSTRACT

As computing evolve, new ways of developing are discussed in a way that sustainability be always present and increasingly improved. Cloud computing evolved to a point that it is able to offer many resources for systems implementation in a sustainable, scalable and accessible way for everyone who wants, besides a low cost for the initial investment. The objective of this work it's develop a *mobile* application with all its *backend* processing run in the cloud, keeping the reliability, low latency and scalability, witch it will grant a great user experience. Serverless architecture methods and access control in the cloud it will be presented along with technologies like non relational databases and code execution without server provisioning. The proposed system will be implanted in the AWS cloud environment.

**Keywords:** Cloud Computing. Serverless. IaaS. Microservices



## LISTA DE ILUSTRAÇÕES

Figura 1 – IaaS, PaaS e SaaS (AZURE, 2019c). Adaptado . . . . .	16
Figura 2 – Aplicação Monolítica X Microserviços, (FOWLER; LEWIS, 2014). Adaptado . . . . .	20
Figura 3 – Representação de itens em tabelas do <i>DynamoDB</i> , (AWS, 2019d). Adaptado. . . . .	25
Figura 4 – Arquitetura simplificada do sistema . . . . .	28
Figura 5 – Criação do usuário IAM e tipos de acesso do usuário IAM . . . . .	30
Figura 6 – Anexação de políticas de permissão ao usuário IAM . . . . .	31
Figura 7 – Criação da tabela eventos . . . . .	32
Figura 8 – Teste de método <i>get-usuario</i> via console AWS . . . . .	36
Figura 9 – Teste de método <i>get-usuario</i> via <i>Postman</i> . . . . .	37
Figura 10 – Esquema de acessos da função <i>getUsuario</i> visualizado através do console AWS . . . . .	38
Figura 11 – Tela de cadastro . . . . .	49
Figura 12 – Tela de login . . . . .	49
Figura 13 – Tela de agenda . . . . .	50
Figura 14 – Tela de listagem de eventos . . . . .	50
Figura 15 – Tela de criação de eventos . . . . .	50
Figura 16 – Tela de listagem de contatos . . . . .	50
Figura 17 – Tela criação de novo contato . . . . .	51
Figura 18 – Tela de listagem de convites . . . . .	51
Figura 19 – Tela de visualização de evento . . . . .	51
Figura 20 – Tela de visualização de convite . . . . .	51

## LISTA DE TABELAS

Tabela 1 – Microserviços do sistema . . . . .	44
Tabela 2 – Tabelas <i>DynamoDB</i> . . . . .	45
Tabela 3 – Telas da Interface de usuário . . . . .	46

## LISTA DE ABREVIATURAS E SIGLAS

<b>AWS</b>	<i>Amazon Web Services</i> .....	<b>13</b>
<b>API</b>	<i>Application Programming Interface</i> .....	<b>14</b>
<b>MFA</b>	<i>Autenticação Multifator</i> .....	<b>23</b>
<b>BI</b>	<i>Business Intelligence</i> .....	<b>17</b>
<b>HTTP</b>	<i>Hypertext Transfer Protocol</i> .....	<b>14</b>
<b>IaaS</b>	<i>Infrastructure as a Service</i> ou <i>Infraestrutura como serviço</i> .....	<b>13</b>
<b>IAM</b>	<i>Identity Access Manager</i> .....	<b>14</b>
<b>JSON</b>	<i>Javascript Object Notation</i> .....	<b>24</b>
<b>NoSQL</b>	<i>Not Only SQL</i> .....	<b>21</b>
<b>PaaS</b>	<i>Platform as a Service</i> ou <i>Plataforma como serviço</i> .....	<b>16</b>
<b>PITR</b>	<i>Recuperação point-in-time</i> .....	<b>24</b>
<b>RDS</b>	<i>Relational Database Service</i> .....	<b>40</b>
<b>REST</b>	<i>Representational State Transfer</i> .....	<b>22</b>
<b>SaaS</b>	<i>Software as a Service</i> ou <i>Software como serviço</i> .....	<b>16</b>
<b>SDK</b>	<i>Software Development Kit</i> .....	<b>23</b>
<b>SES</b>	<i>Simple E-mail Service</i> .....	<b>40</b>
<b>SNS</b>	<i>Simple Notification Service</i> .....	<b>40</b>
<b>SQL</b>	<i>Structured Query Language</i> .....	<b>21</b>
<b>SSD</b>	<i>Discos de Estado Sólido</i> .....	<b>24</b>
<b>TI</b>	<i>Tecnologia da Informação</i> .....	<b>16</b>

# SUMÁRIO

<b>1</b>	<b>INTRODUÇÃO</b>	<b>13</b>
<b>1.1</b>	<b>OBJETIVOS</b>	<b>14</b>
1.1.1	Objetivos Gerais	14
1.1.2	Objetivos Específicos	15
<b>2</b>	<b>FUNDAMENTAÇÃO TEÓRICA</b>	<b>16</b>
<b>2.1</b>	<b>SERVIÇOS BASEADOS NA COMPUTAÇÃO EM NUVEM</b>	<b>16</b>
2.1.1	IaaS	16
2.1.2	PaaS	17
2.1.3	SaaS	18
<b>2.2</b>	<b>ARQUITETURA</b>	<b>18</b>
2.2.1	Arquitetura de Microsserviços	18
<b>2.3</b>	<b>NOSQL</b>	<b>21</b>
<b>3</b>	<b>TECNOLOGIAS E MÉTODOS</b>	<b>22</b>
<b>3.1</b>	<b>AWS</b>	<b>22</b>
3.1.1	Amazon API Gateway	22
3.1.2	Lambda	22
3.1.3	Amazon Cognito	23
3.1.4	IAM	24
3.1.5	Amazon Cloud Watch	24
3.1.6	DynamoDB	24
3.1.7	Serverless Framework	26
3.1.8	AWS Amplify	26
<b>3.2</b>	<b>REACT NATIVE</b>	<b>27</b>
<b>4</b>	<b>DESENVOLVIMENTO</b>	<b>28</b>
<b>4.1</b>	<b>ARQUITETURA DO SISTEMA</b>	<b>28</b>
4.1.1	Autenticação	28
4.1.2	Comunicação com a API	29
4.1.3	Processamento	29
4.1.4	Armazenamento	29

<b>4.2</b>	<b>DESENVOLVIMENTO DO BACKEND</b>	<b>30</b>
4.2.1	Usuário do IAM	30
4.2.1.1	Chaves de acesso	31
4.2.2	User pool	31
4.2.2.1	Adicionar uma aplicação cliente	32
4.2.3	Tabelas do <i>DynamoDB</i>	32
4.2.4	Codificação das APIs	33
4.2.5	Identity Pool	33
<b>4.3</b>	<b>INTERFACE DE USUÁRIO</b>	<b>34</b>
4.3.1	AWS Amplify	34
<b>5</b>	<b>RESULTADOS</b>	<b>36</b>
<b>5.1</b>	<b>API GATEWAY</b>	<b>36</b>
<b>5.2</b>	<b>LAMBDA</b>	<b>37</b>
<b>5.3</b>	<b>DYNAMODB</b>	<b>37</b>
<b>6</b>	<b>CONCLUSÃO</b>	<b>39</b>
6.0.1	Vantagens	39
6.0.2	Desvantagens	39
<b>6.1</b>	<b>TRABALHOS FUTUROS</b>	<b>39</b>
	<b>REFERÊNCIAS</b>	<b>41</b>
	<b>APÊNDICES</b>	<b>43</b>
	<b>APÊNDICE A – RELAÇÃO DE MICROSERVIÇOS DO BACKEND</b>	<b>44</b>
	<b>APÊNDICE B – RELAÇÃO DE TABELAS NO DYNAMODB</b>	<b>45</b>
	<b>APÊNDICE C – RELAÇÃO DE TELAS DA INTERFACE DE USUÁRIO</b>	<b>46</b>
	<b>APÊNDICE D – EXEMPLO DE ARQUIVO SERVERLESS.YML PARA CONFIGURAÇÃO DE APIS</b>	<b>47</b>
	<b>APÊNDICE E – TELAS DO APLICATIVO MOBILE</b>	<b>49</b>

## 1 INTRODUÇÃO

A computação em nuvem vem crescendo cada vez mais nos últimos anos e ocupando um espaço importante no mercado de desenvolvimento de sistemas para a *internet*. As expectativas são tamanhas que grandes nomes da computação e informação, tais como *Google*, *Amazon* e *Microsoft* já voltaram suas atenções para tais tecnologias e oferecem serviços tanto implantados como para implantação na nuvem.

Com esse grande crescimento da computação em nuvem surgiram métodos que ajudam o desenvolvedor a implementar e gerenciar aplicações de dimensões e complexidades diferentes. Conceitos como computação *serverless* e *Infrastructure as a Service* ou Infraestrutura como serviço (**IaaS**) ajudam no desempenho da codificação da aplicação e reduz a complexidade da implantação e gerenciamento se comparados aos cenários fora da nuvem (GIENOW, 2018).

Para o desenvolvedor existem diversas vantagens de se implantar sistemas na nuvem. Algumas dessas principais vantagens são apontadas pela Amazon *Web Services* (**AWS**) como:

- a) **Facilidade**: toda a configuração pode ser feita online através do console AWS, a plataforma de gerenciamento da AWS. A empresa ainda oferece o *Amazon Lightsail*, um serviço que apresenta uma interface amigável para facilitar ainda mais o processo de implantação de sistemas na nuvem.
- b) **Flexibilidade**: as ferramentas, linguagens de programação, sistemas operacionais e bancos de dados utilizados ficam todos a critério do desenvolvedor, já que a plataforma de nuvem da AWS fornece suporte a diversos tipos de tecnologia.
- c) **Confiabilidade**: a AWS possui uma infraestrutura global e escalável com redundância e *backups* automáticos, o que garante aos usuários uma aplicação confiável e segura.
- d) **Escalabilidade e desempenho**: são oferecidos recursos de escalabilidade automática e balanceamento de carga para garantir que a aplicação responda rapidamente às requisições e esteja sempre disponível aos usuários.
- e) **Segurança**: a AWS garante segurança aos seus clientes e suas aplicações através de criptografia, registros de *logs*, *firewalls*, identidades, controle de acesso e vários outros recursos de segurança que o desenvolvedor não precisará se preocupar em desenvolver.
- f) **Economia**: uma das vantagens mais marcantes e convenientes ao desenvolvedor é o fato de não haver necessidade de provisionar recursos de *hardware* como investimento inicial, podendo-se simplesmente mensurar quantos e quais recursos a aplicação demanda e a empresa que oferece o serviço de nuvem irá fornecê-los a um preço acessível.

Em termos financeiros, o custo para manter a aplicação também é mais baixo se comparado aos sistemas tradicionais pois o cliente, que, no caso, é o desenvolvedor da aplicação, só é cobrado pelo tempo em que a aplicação esteve em execução, ao passo que nas arquiteturas tradicionais há diversos custos essenciais e vitalícios, tais como energia elétrica, *internet*, mão de obra e outros; além da total responsabilidade sobre possíveis imprevistos e acidentes.

Diante dessa mudança de paradigma, profissionais e pesquisadores se questionam sobre a transição dos tradicionais sistemas monolíticos para esse novo conceito de arquitetura distribuída em microserviços, bem como a implantação *serverless*. Sabendo dessas questões acerca do desenvolvimento de sistemas fazendo uso dos novos conceitos de computação em nuvem, este trabalho propõe uma aplicação *mobile* com todo o seu processamento, armazenamento e autenticação rodando na nuvem através dos serviços fornecidos pela **AWS** a fim de demonstrar e discutir a implementação de um sistema *serverless* na nuvem.

A aplicação proposta faz uso do serviço de autenticação *Amazon Cognito*, bem como funções *Lambda* para processamento, *DynamoDB* como banco de dados não relacional para armazenamento e persistência de dados, *Amazon Identity Access Manager (IAM)* para controle de acessos e *Amazon API Gateway* para orquestramento das requisições *Hypertext Transfer Protocol (HTTP)* feitas ao *backend*. Todos os serviços citados são oferecidos pela **AWS**. O *frontend*, por sua vez, será um aplicativo multiplataforma a ser desenvolvido através do *framework React Native*.

Para construção do ambiente na nuvem foi utilizado o *framework Serverless*. Essa ferramenta fornece recursos para desenvolvimento de *Application Programming Interface (API)*s altamente escaláveis na nuvem. Oferece suporte de desenvolvimento a diversos ambientes em nuvem, tais como *AWS Cloud*, *Microsoft Azure* e outros.

Por fim foi desenvolvido uma aplicação *mobile* multiplataforma para utilização dos usuários utilizando o *framework React Native*. A conexão entre *frontend* e *backend* foi feita utilizando o *AWS Amplify*, uma biblioteca oferecida pela **AWS** para desenvolvimento de aplicações *web* e *mobile* integradas com *backends* implantados em ambiente de nuvem da **AWS**.

## 1.1 OBJETIVOS

Os objetivos dessa monografia foram divididos entre gerais e específicos.

### 1.1.1 Objetivos Gerais

O objetivo deste trabalho é estudar e desenvolver um sistema cujo *backend* é integralmente implantado em nuvem de forma segura, confiável, disponível, com baixa latência, utilizando arquitetura de um sistema orientado a funções e serviços oferecidos pela **AWS**.

O projeto proposto é um aplicativo *mobile* multiplataforma capaz de programar reuniões corporativas envolvendo vários usuários.

### 1.1.2 Objetivos Específicos

- O sistema deve permitir autenticação segura e consistente independentemente do dispositivo utilizado pelo usuário.
- Controle de acesso dos usuários aos recursos do ambiente na nuvem.
- Armazenamento e resgate de dados de forma confiável e rápida utilizando um banco de dados não relacional.
- Utilização de arquitetura orientada a funções.
- Processamento feito em nuvem sem provisionamento de *hardware* por parte do desenvolvedor.
- Implantação automatizada do sistema no ambiente de nuvem.



## 2 FUNDAMENTAÇÃO TEÓRICA

A **AWS** define computação em nuvem como “a entrega sob demanda de poder computacional, armazenamento de banco de dados, aplicativos e outros recursos de Tecnologia da Informação (**TI**) pela *internet* com uma definição de preço conforme o uso”. Sendo possível entregar poder de processamento e outros recursos conforme o uso, uma série de benefícios podem surgir tanto para os pequenos como para os grandes desenvolvedores.

Grandes empresas, tais como *Google*, *Microsoft* e *Amazon*, enxergaram uma oportunidade de mercado nessa nova ideia de entrega de recursos sob demanda e começaram a investir no assunto. Algumas dessas empresas chegaram a um alto nível de refinamento dos serviços de computação em nuvem, tornaram-se referência no assunto e começaram a fazer disso um negócio que vem transformando o desenvolvimento e a implantação de aplicações na *internet*.

### 2.1 SERVIÇOS BASEADOS NA COMPUTAÇÃO EM NUVEM

Com o mercado de *software* e infraestrutura aquecido devido ao investimento das grandes empresas nos serviços de entrega sob demanda baseados em nuvem, novos termos surgiram para definir o que são esses serviços. São eles: **IaaS**, *Platform as a Service* ou Plataforma como serviço (**PaaS**) e *Software as a Service* ou Software como serviço (**SaaS**), Figura 1.

#### 2.1.1 IaaS

Trata-se de fornecer recursos computacionais como serviço a um cliente que deseja implantar uma aplicação.

O desenvolvedor pode definir qual o poder de processamento sua aplicação demanda



**Figura 1** – IaaS, PaaS e SaaS (**AZURE**, 2019c). Adaptado

e a empresa que oferece o serviço tomará as providências para fornecê-lo, seja um recurso virtual ou um *hardware* dedicado. O provedor de serviços gerencia o *hardware* e o *software* demandado pela aplicação e ainda é capaz de garantir a segurança, persistência de dados, confiabilidade e baixa latência (AZURE, 2019a).

A AWS possui *datacenters* em várias localidades do mundo, sendo possível iniciar várias instâncias da aplicação em diferentes pontos geográficos a fim de atender clientes de diferentes regiões com a menor latência possível. Tais recursos são facilmente escaláveis tanto horizontal quanto verticalmente, ou seja, os recursos são mais flexíveis que no modelo de implantação tradicional (AWS, 2019m).

Se compararmos com os métodos tradicionais de hospedagem e implantação, as diferenças são grandes, especialmente nos aspectos financeiros. Para o desenvolvedor que está começando um novo negócio baseado no modelo SaaS, providenciar recursos para suportar um sistema que ainda está sendo desenvolvido pode ser bastante penoso e especulativo, de forma que o servidor provisionado pode possuir um poder de processamento abaixo daquele realmente demandado pela aplicação, gerando assim problemas de latência ou até mesmo perda de informação, como também pode possuir poder de processamento muito acima do exigido pela aplicação, o que acarretará em ociosidade de recursos que poderiam ter sido economizados no orçamento.

Em um modelo *pay-per-use* (pague por uso) e com escalabilidade flexível esses problemas não ocorrem, o que gera uma grande economia ao desenvolvedor que está começando seu negócio, tornando mais viável e conveniente implantar seu *software* em um servidor baseado em nuvem com entrega de recursos sob demanda.

### 2.1.2 PaaS

São plataformas de gerenciamento fornecidas pela empresa provedora dos recursos de infraestrutura na nuvem. Através do PaaS o próprio desenvolvedor pode executar todo tipo de tarefas que sejam importantes para o ciclo de vida da sua aplicação, tais como testes, provisionamento de recursos extra nos horários de maior carga de requisições, controle de acessos dentre outras.

O PaaS engloba o conceito de IaaS por que é através da plataforma que o desenvolvedor dará suporte à sua aplicação e administrará todos os recursos de infraestrutura que a mantém funcionando. Além dos recursos básicos de infraestrutura, frequentemente encontramos, em ferramentas PaaS, recursos de *Business Intelligence* (BI) que auxiliam o desenvolvedor a tomar decisões importantes para alinhamento tanto de projeto como de sua empresa (AZURE, 2019b).

Plataformas PaaS são comumente acessadas pelo navegador de *internet* e possuem interfaces amigáveis, economizando tempo, dinheiro e esforço dos responsáveis pela aplicação.

### 2.1.3 SaaS

Trata-se de um modelo de licenciamento e entrega de *softwares* que são hospedados no servidor de uma empresa que forneça o serviço de infraestrutura, o *software* chega aos clientes via *internet* (AWS, 2019l). Os clientes podem realizar uma assinatura do *software*, e acessá-lo pela rede através do navegador em qualquer dispositivo, uma vez que não há necessidade de instalar um programa para utilizar o serviço. Alguns exemplos de *softwares* como serviço são aplicativos de *e-mail* baseados na *web*, agendas e calendários (AZURE, 2019c).

Atualmente existe uma série de ferramentas que facilitam e automatizam o processo de *deploy* de aplicações modelo SaaS em servidores, o que deixa a implantação mais rápida. Um exemplo de ferramenta que auxilia nesse processo é o *Docker*, que permite uma completa customização do ambiente a ser implantado, centralizando os recursos e dependências da aplicação em uma estrutura de contêineres que otimiza a execução da aplicação em diferentes ambientes, além de ocupar menos espaço em *hardware* (AWS, 2019k).

O SaaS também possui a vantagem de ser um modelo *pay-per-use* e facilita a ampliação do público alvo da empresa a um nível global, já que o sistema implantado pode ser oferecido em qualquer parte do mundo (AWS, 2019l).

O modelo SaaS engloba também os conceitos de IaaS e PaaS, afinal, o SaaS é o produto final que chega até o cliente.

## 2.2 ARQUITETURA

Arquitetura de um sistema é a forma como o sistema é organizado e estruturado. Também envolve como as partes do sistema se comunicam.

Os sistemas tradicionais possuem uma arquitetura monolítica, onde o servidor é composto por um bloco único. Geralmente as aplicações tradicionais que adotam uma arquitetura monolítica são divididas em três grandes partes: uma interface de usuário (*frontend*), um servidor (*backend*) e um banco de dados (FOWLER; LEWIS, 2014).

Nessa estrutura, o servidor recebe requisições, comumente requisições HTTP, e as manipula a fim de executar uma lógica do sistema. Dessa forma, são realizadas operações no banco de dados, e ao concluir a tarefa, o servidor envia uma resposta à interface de usuário de acordo com os resultados da operação. (FOWLER; LEWIS, 2014).

Apesar de a arquitetura monolítica ser a mais comumente utilizada, existe um novo modelo para se construir sistemas que vem sido bastante explorada nos últimos anos.

### 2.2.1 Arquitetura de Microserviços

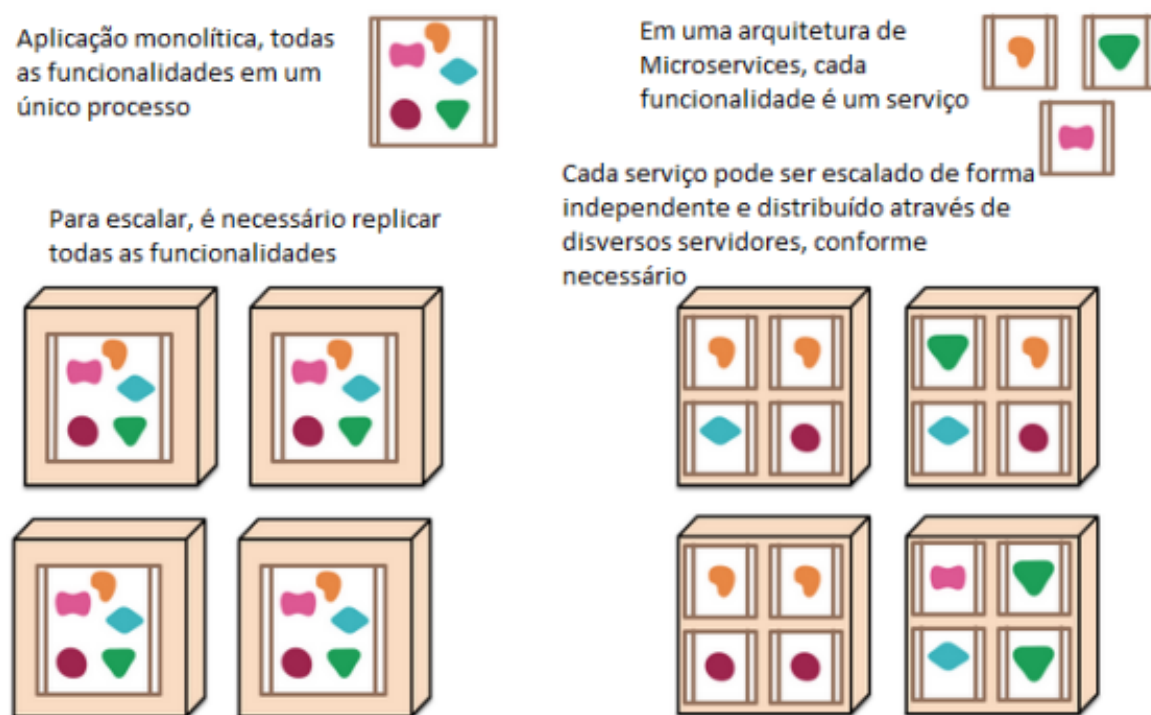
A arquitetura de microserviços é uma abordagem para desenvolvimento de uma única aplicação como um conjunto de pequenos serviços rodando seus próprios processos e se comuni-

cando através de mecanismos leves e rápidos. Não se trata de um modelo vertical e centralizado como o anterior e seus componentes (serviços) podem ser implantados independentemente e o processo de implantação é facilmente automatizado (FOWLER; LEWIS, 2014).

Uma estrutura bastante comum em sistemas baseados em microserviços é composta por uma API que recebe requisições HTTP e dispara funções específicas atreladas a rotas bem definidas de acordo com o que a requisição pede. A biblioteca *Flask* utiliza essa ideia para o desenvolvimento de aplicações baseadas em microserviços utilizando a linguagem de programação *Python*. Já a AWS oferece o recurso denominado *API Gateway*, que segue o mesmo princípio ao disparar funções Lambda de acordo com a rota especificada na requisição.

A arquitetura de microserviços (também conhecida como arquitetura orientada a serviços ou arquitetura orientada a funções) vem ganhando bastante espaço no mercado, tanto nas grandes quanto nas pequenas empresas. Isso se deve às suas características, benefícios e praticidades:

- a) **Autonomia:** Cada componente pode ser desenvolvido, implantado, operado e escalado individualmente sem afetar os demais. Não compartilham código com outros serviços e se comunicam através de APIs bem definidas (AWS, 2019j).
- b) **Especialização:** Cada microserviço é designado a uma tarefa única, o que deixa seu código fonte menor, mais fácil de se compreender e desenvolver. Como consequência disso, sistemas construídos nessa arquitetura possuem uma menor complexidade em relação aos sistemas monolíticos, pois estes possuem todos os serviços concentrados em um único processo e tendem a ficar mais complexos à medida que são incorporadas novas funcionalidades, o que dificulta a manutenção do sistema (AWS, 2019j).
- c) **Agilidade:** Uma empresa pode ter vários times, onde cada um trabalha desenvolvendo e mantendo um único serviço. Assim as equipes trabalham de forma independente e com maior eficiência e organização (AWS, 2019j).
- d) **Escalabilidade Flexível:** Os serviços podem ser escalados de forma independente e flexível tanto horizontal quanto verticalmente, como mostra a Figura 2, diferentemente da arquitetura monolítica, onde a escalabilidade depende da aquisição de novas máquinas. Assim, é possível prover recursos para atender exatamente o que cada serviço demanda, mesmo em horários de picos de demanda (AWS, 2019j).
- e) **Fácil Implantação:** Uma vez que podem ser derrubados e implantados de forma automatizada e sem afetar os demais serviços, é mais viável realizar testes de novas funcionalidades ou de correções na aplicação, já que o custo e risco são mais baixos que em um sistema monolítico, onde para implantar uma nova funcionalidade é necessário parar todo o sistema e implantá-lo novamente. Esse processo costuma ser feito fora do expediente dos funcionários por serem horários com baixa demanda,



**Figura 2** – Aplicação Monolítica X Microserviços, (FOWLER; LEWIS, 2014). Adaptado

o que é custoso tanto para os funcionários como para a empresa.

- f) **Variedade de tecnologias:** Por serem independentes, os serviços podem ser feitos utilizando tecnologias distintas, o que abre ao desenvolvedor uma gama de possibilidades, dado que ferramentas diferentes são adequadas para tarefas diferentes. Assim, um microserviço relacionado a processamento gráfico pode ser desenvolvido em C++ (linguagem de programação comumente utilizada para esse tipo de tarefa), enquanto outro que faz análise de dados pode ser desenvolvido em Python (que possui diversas ferramentas utilizadas em *Data Analysis*).
- g) **Reutilização de código:** Ao dividir o sistema em pequenas partes é possível chamar serviços já implementados dentro de outros serviços, reaproveitando código já desenvolvido, o que também contribui para a agilidade do desenvolvimento (AWS, 2019j).
- h) **Resiliência:** Devido à independência dos serviços, a aplicação é mais resistente a falhas, diferente da arquitetura monolítica, onde a falha de um único serviço pode comprometer todo o sistema. Para manutenção, só é preciso parar o serviço com defeito enquanto o restante da aplicação continua a funcionar (FOWLER; LEWIS, 2014).

## 2.3 NOSQL

O Bancos de dados tradicionais, chamados relacionais, são compostos por tabelas que mapeiam entidades fortemente relacionadas entre si através de um esquema de dados. Assim oferecem recursos de relacionamento extremamente confiáveis e mantêm o registro de dados consistente. Entretanto, tamanho cuidado na consistência e confiabilidade da informação é negativamente refletido no desempenho do banco, que, dependendo da aplicação, fica comprometido. O custo de latência para administrar esses relacionamentos pode ser alto demais caso a aplicação precise de uma resposta rápida e também pode levar a uma sobrecarga até mesmo em tarefas relativamente simples (JUNIOR et al., 2018).

Assim, em 1998 surgiu pela primeira vez o termo *Not Only SQL* (NoSQL), na apresentação do Strozzi NoSQL, que, introduzido por Strozzi, era um banco de dados relacional que não utilizava o *Structured Query Language* (SQL), linguagem de consulta utilizada nos bancos de dados tradicionais (TAMANE, 2016). Posteriormente, em 2009, o termo foi utilizado novamente por Johan Oskarsdon, na conferência conhecida como "*NoSQL Meetup*", porém, dessa vez, já referenciando bancos de dados não relacionais (CATTELL, 2011).

Os bancos de dados conhecidos como não relacionais, ou NoSQL vêm sendo cada vez mais utilizados para suprir essas dificuldades de desempenho. Foram projetados para lidar com grandes quantidades de dados e armazená-los de forma distribuída e não possuem um tipo fixo como os esquemas definidos nos bancos de dados tradicionais (JUNIOR et al., 2018).

Entretanto, manter a confiabilidade e consistência da informação é um ponto crucial para a TI, de forma que os bancos de dados não relacionais não vieram para extinguir os bancos de dados relacionais ou a linguagem SQL, mas sim para que os dois tipos de bancos de dados sejam usados de forma combinada, para oferecer respostas rápidas, consistentes e estruturadas (ABRAMOVA; BERNARDINO, 2013).

Alguns bancos de dados não relacionais já existentes possuem fundamentos diferentes entre si, o que leva o desenvolvedor a estudar a fundo o banco de dados escolhido para o seu projeto, e esses diferentes fundamentos implicam em diferentes formas de suprir as dificuldades encontradas nos bancos NoSQL, principalmente as atreladas à ausência do relacionamento e referenciamento. No decorrer deste trabalho serão apresentados alguns conceitos e fundamentos utilizados pelo *DynamoDB*, serviço de banco de dados não relacional da AWS, para lidar grandes quantidades de dados distribuídos e não relacionados.

### 3 TECNOLOGIAS E MÉTODOS

Esse capítulo descreve todas as tecnologias utilizadas na implementação e implantação da aplicação *serverless* na nuvem. Também são apresentados recursos dedicados à manutenção e gerenciamento do sistema que podem ser usados tanto em ambiente de desenvolvimento como de produção.

#### 3.1 AWS

*Amazon Web Services*, ou simplesmente **AWS** é um conjunto de ferramentas e serviços oferecidos pela empresa *Amazon.com* voltados para desenvolvimento e manutenção de sistemas em nuvem. Atualmente, a **AWS** é reconhecida pela qualidade dos seus serviços e alcançou um grande espaço no mercado de *software* pela sua excelência em serviços de nuvem, sendo assim uma das plataformas mais utilizadas para se trabalhar com *cloud computing* no mundo (**AWS**, 2019i).

A plataforma de Nuvem AWS possui mais de 165 serviços distintos, incluindo cerca de 40 serviços que não são oferecidos por outras empresas do segmento (**AWS**, 2019i). Alguns exemplos são processamento remoto, virtualização de ambientes em nuvem, controle de acesso, autenticação, inteligência artificial, armazenamento de dados e arquivos, dentre outros.

A seguir serão expostos os serviços da AWS que foram utilizados na construção deste trabalho.

##### 3.1.1 Amazon API Gateway

O *Amazon API Gateway* é um serviço da AWS que permite criar, publicar e gerir APIs que sigam tanto baseadas em *Representational State Transfer* (**REST**) quanto em *WebSocket*. Através desse serviço é possível criar **APIs** que acessem outros serviços **AWS** ou serviços externos da *web* (**AWS**, 2019a). O modelo de **API** escolhido para este trabalho foi o **REST**.

O **REST** (*Representational State Transfer*) é uma representação abstrata da arquitetura *web* que define regras para transferência de informações pela rede. Utilizando **REST** uma **API** pode ser consumida por aplicações implantadas em diferentes plataformas via métodos **HTTP** (GET, POST, PUT, PATCH e DELETE) (**PIRES**, 2017).

##### 3.1.2 Lambda

O **AWS** *Lambda* é um serviço de computação oferecido pela AWS que permite execução de códigos na nuvem com cobrança sob demanda, sem provisionamento de servidores e escalabilidade automática. O serviço oferece alta disponibilidade e realiza automaticamente

tarefas relacionadas manutenção de servidor e sistemas operacionais, provisionamento de capacidade de processamento e até mesmo monitoramento de código e registro de *logs* de operações. Além disso são varias as linguagens de programação compatíveis com o serviço e que podem ser utilizadas (AWS, 2019h).

Os trechos de código executados pelo serviço são chamados de funções *Lambda*, e podem ser chamados em respostas a eventos disparados por outros serviços da AWS, por requisições HTTP intermediadas pelo API Gateway; ou mesmo diretamente da interface de usuário, utilizando um *Software Development Kit* (SDK) oferecido pela AWS (AWS, 2019h).

Na aplicação aqui proposta, funções *Lambda* serão executadas através de requisições HTTP feitas pelo cliente através de um smartphone, e que serão gerenciadas pelo API Gateway a fim de padronizar as chamadas de API.

### 3.1.3 Amazon Cognito

O *Amazon Cognito* é um serviço de autenticação e gerenciamento de usuários fornecido pela AWS. Utilizando o serviço, usuários da aplicação podem efetuar *login* através de *e-mail* e senha ou através de contas em redes sociais, tais como *Facebook* e *Google*. Os principais componentes do *Amazon Cognito* são os grupos de usuários e os grupos de identidades, que podem ser utilizados separadamente ou em conjunto (AWS, 2019c).

- *Grupos de Usuários* são diretórios de usuários que fornecem opções de cadastro e *login* para os usuários da aplicação. Com os grupos de usuários, cada usuário possui um perfil de diretório, o que garante uma experiência de usuário consistente, independentemente do dispositivo utilizado pelo cliente naquele momento. Além disso, os grupos de usuários fornecem recursos de segurança como a Autenticação Multifator (MFA), que verifica a existência de credenciais comprometidas, proteção de aquisição de conta e verificação de *e-mail* e telefone (AWS, 2019c).
- Os *Grupos de Identidades* permitem a criação de identidades para os usuários e os autentica com provedores de identidade federada. Com uma identidade federada, é possível obter credenciais temporárias, com privilégios limitados para acessar com segurança outros serviços da AWS, tais como *Amazon DynamoDB*, e *Amazon API Gateway*, utilizados na aplicação (AWS, 2019c).

Ao efetuar com sucesso um *login* no aplicativo através de um grupo de usuários (que garantirá consistência e segurança das informações do usuário), o *Amazon Cognito* fornece um *token* ao cliente, que está atrelado a uma identidade federada (em um grupo de identidades). Essa identidade contém uma relação de serviços e recursos que o usuário, agora autenticado, poderá acessar temporariamente.



### 3.1.4 IAM

O [IAM](#) é o serviço utilizado para fornecer, de forma segura, acessos aos recursos da [AWS](#) utilizados pela aplicação. É usado para controlar a autenticação dos usuários e para controlar os acessos de cada um. Uma empresa que assina a plataforma de nuvem da [AWS](#) pode utilizar o [IAM](#) para criar diferentes perfis de acesso de acordo com as ocupações dos funcionários. Aos empregados, por sua vez, são atribuídos os perfis criados de forma que cada funcionário consiga manipular apenas os recursos que lhe competem ([AWS](#), 2019g).

Outro recurso interessante do [IAM](#) é a criação de *roles*, ou funções do [IAM](#). Uma função do [IAM](#) é uma relação de acessos que podem ser atribuídas a um determinado serviço [AWS](#). Ao atribuir uma *role* a uma função *Lambda*, por exemplo, pode-se especificar com quais serviços a função *Lambda* poderá interagir e quais as ações a função poderá executar sobre aquele serviço, como, por exemplo, registrar um log no *Amazon Cloud Watch*.

### 3.1.5 Amazon Cloud Watch

Serviço que monitora os demais recursos da [AWS](#) em tempo real. O *Amazon Cloud Watch* mantém registradas todas as informações relacionada às execuções de funções *lambda* e operações realizadas por outros serviços [AWS](#) ([AWS](#), 2019b). Na realização deste trabalho foi fundamental na geração de *logs* para desenvolvimento, *deploy* e testes do *backend* da aplicação e na comunicação entre o *frontend* e o *API Gateway*.

### 3.1.6 DynamoDB

O *DynamoDB* é um serviço de bancos de dados distribuídos [NoSQL](#) utilizado para armazenamento e persistência de dados na nuvem da [AWS](#). Os dados armazenados pelo *DynamoDB* são salvos em Discos de Estado Sólido ([SSD](#)) e replicados em diversas zonas de disponibilidade em uma região da [AWS](#), o que garante tanto uma alta disponibilidade e confiança quanto uma baixa latência na troca de informações ([AWS](#), 2019d).

Bem como os demais serviços da [AWS](#), o *DynamoDB* é automaticamente escalável. Também conta com criptografia em repouso, o que deixa a responsabilidade da segurança das informações armazenadas com a [AWS](#), e não com o desenvolvedor. Além disso é possível realizar *backups* sob demanda das tabelas Recuperação *point-in-time* ([PITR](#)), o que protege o banco de possíveis operações acidentais dentro dos últimos 35 dias ([AWS](#), 2019d).

No *DynamoDB*, os dados são organizados em tabelas, itens e atributos. Uma tabela é uma coleção de itens, um item é uma coleção de atributos identificado por uma chave primária. É uma estrutura de certa forma semelhante à dos bancos de dados relacionais tradicionais. Entretanto, o *DynamoDB* não possui esquemas, o que permite que cada item tenha atributos diferentes e únicos. A estrutura de armazenamento das tabelas pode ser comparada à estrutura de um *Javascript Object Notation* ([JSON](#)) como fica evidente na Figura 3a ([AWS](#), 2019e).

People	Music
<pre>{   "PersonID": 101,   "LastName": "Smith",   "FirstName": "Fred",   "Phone": "555-4321" }</pre>	<pre>{   "Artist": "No One You Know",   "SongTitle": "My Dog Spot",   "AlbumTitle": "Hey Now",   "Price": 1.98,   "Genre": "Country",   "CriticRating": 8.4 }</pre>
<pre>{   "PersonID": 102,   "LastName": "Jones",   "FirstName": "Mary",   "Address": {     "Street": "123 Main",     "City": "Anytown",     "State": "OH",     "ZIPCode": 12345   } }</pre>	<pre>{   "Artist": "No One You Know",   "SongTitle": "Somewhere Down The Road",   "AlbumTitle": "Somewhat Famous",   "Genre": "Country",   "CriticRating": 8.4,   "Year": 1984 }</pre>
<pre>{   "PersonID": 103,   "LastName": "Stephens",   "FirstName": "Howard",   "Address": {     "Street": "123 Main",     "City": "London",     "PostalCode": "ER3 5K8"   },   "FavoriteColor": "Blue" }</pre>	<pre>{   "Artist": "The Acme Band",   "SongTitle": "Still in Love",   "AlbumTitle": "The Buck Starts Here",   "Price": 2.47,   "Genre": "Rock",   "PromotionInfo": {     "RadioStationsPlaying": [       "KHCR",       "KQBX",       "WTNR",       "WJHJ"     ],     "TourDates": {       "Seattle": "20150625",       "Cleveland": "20150630"     },     "Rotation": "Heavy"   } }</pre>

(a) Tabela *People* com apenas chave de partição (b) Tabela *Music* com chave de partição e chave de classificação

**Figura 3** – Representação de itens em tabelas do *DynamoDB*, (AWS, 2019d). Adaptado.

Na tabela *People* vista na figura 3a, a chave primária é o atributo *PersonID*. Esse é o único atributo obrigatório para a inserção de um novo item na tabela. Todos os demais atributos são dispensáveis, e podem ser variados.

Entretanto, há dois tipos de chaves primárias que podem ser utilizadas no *DynamoDB*, as chaves de partição e as chaves de classificação.

- **Chaves de partição** são chaves simples compostas por apenas um atributo como identificador único. Internamente a chave de partição é usada como identificador para uma função de *hash*, a saída dessa função *hash* indica o local do armazenamento físico daquele item (AWS, 2019d).

A tabela *People* é um exemplo de tabela com chave de partição. Esse tipo de chave permite a busca de um item específico na tabela fornecendo apenas o valor de sua chave.

- Já as **Chave de partição e chave de classificação** são compostas por dois atributos e por essa razão são conhecidas como chaves primárias compostas. O primeiro atributo é conhecido como chave de partição e o segundo como chave de classificação. Devido ao seu uso interno, a chave de partição de um item também é conhecida como seu atributo de *hash*

(AWS, 2019d).

A chave de partição é utilizada como entrada para a função *hash* interna do *DynamoDB* assim como no item anterior, definindo a posição física onde o item será armazenado. Porém, com a chave de classificação, todos os itens com a mesma chave de partição são armazenados juntos, em sequência, de acordo com o valor de suas chaves de classificação. Ou seja, em uma tabela com esse tipo de chave, dois itens podem possuir o mesmo valor de chave de partição desde que possuam chaves de classificação diferentes (AWS, 2019d).

Essa estrutura de chaves fornece maior flexibilidade para consultar dados de uma tabela, pois caso seja fornecido o valor de uma chave de partição, todos os itens com aquela chave de partição serão retornados sem que hajam grandes atrasos na consulta, uma vez que esses itens estão fisicamente próximos no banco de dados (AWS, 2019d).

Um exemplo de tabela com uso de chave de partição de chave de classificação é visto tabela *Music*, na Figura 3b. Devido ao seu uso interno como a forma como o *DynamoDB* armazena os dados próximos uns dos outros, a chave de classificação de um item também é conhecida como seu atributo de intervalo (AWS, 2019d).

### 3.1.7 Serverless Framework

O *Serverless framework* é uma ferramenta que abstrai os principais conceitos de computação sem servidor para facilitar a implantação de sistemas *serverless*, o que faz com que o desenvolvedor não tenha que se preocupar com a difícil tarefa de implantar um complexo sistema *serverless*, já que esse processo é abstraído e automatizado pelo *Serverless Framework*.

Dessa forma, a equipe de desenvolvimento passa a maior parte do tempo, de fato, codificando, o que aumenta a eficiência e agilidade do trabalho de implementação e implantação. O *framework* utiliza um arquivo *YAML*, que descreve toda a estrutura da aplicação sem servidor. Uma vez com o arquivo pronto, basta um comando para que se inicie o processo de *deploy* em um provedor de nuvem. Além disso, o *Serverless* permite fazer controles de versão, possibilitando reverter o processo em caso de erro de implantação. O *Serverless* possui suporte para diversas plataformas de nuvem, tais como *AWS Cloud*, *Google Cloud*, *Azure*, dentre outros (SERVERLESS, 2018).

### 3.1.8 AWS Amplify

O *AWS Amplify*, ou simplesmente *Amplify*, é uma biblioteca fornecida pela própria AWS para ser usada em aplicações desenvolvidas para *Android*, *iOS*, *web* (suporta alguns *frameworks Javascript*, como *Angular* e *ReactJS*) e *React Native* integradas com ambientes na nuvem. O *Amplify* oferece recursos e métodos prontos para serem utilizados, tornando fácil a integração do *frontend* com o *backend* na nuvem. Também oferece uma interface de linha de comando que pode ser utilizada para *setup* e *deploy* de recursos de nuvem para o *backend* (AWS, 2019f).

## 3.2 REACT NATIVE

O *React Native* é um módulo do *framework* React, que foi introduzido no mercado através da *Facebook Inc.*, sua desenvolvedora e distribuidora, e teve grande sucesso no desenvolvimento de aplicações *web* devido à sua versatilidade, eficiência e reaproveitamento de código. Pensando em uma ferramenta equivalente para desenvolvimento *mobile* multiplataforma, a empresa também desenvolveu o *React Native*, que permite desenvolvimento de aplicações *mobile* usando elementos fundamentais de interface de usuário nativos das plataformas *Android* e *iOS*. O *React Native* utiliza a mesma estrutura do *React*, o que torna muito mais fácil criar uma solução *web* para o projeto em um trabalho futuro (FACEBOOK, 2017).

Levando em conta seu espaço cada vez mais crescente no mercado, o *React Native* foi escolhido como *framework* para desenvolvimento da aplicação proposta neste trabalho.

## 4 DESENVOLVIMENTO

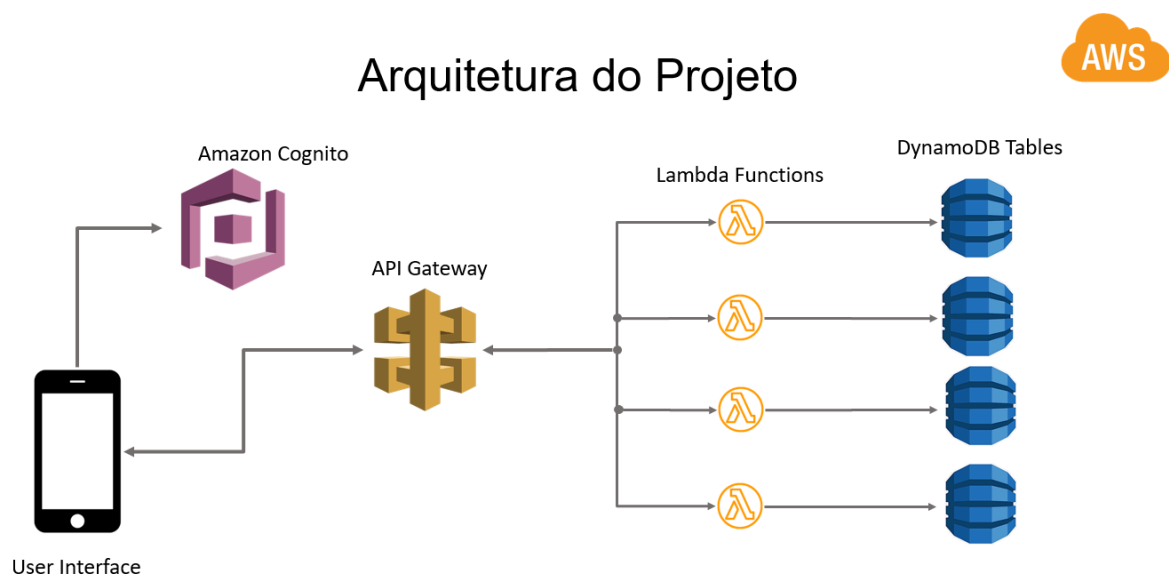
Esse capítulo descreve a organização da aplicação, tanto *frontend* quanto *backend*, de forma um pouco mais detalhada, mostrando as técnicas e decisões de projeto adotadas em termos de processamento, armazenamento, segurança, comunicação e implantação.

### 4.1 ARQUITETURA DO SISTEMA

A arquitetura do sistema é apresentada na Figura 4. Trata-se de uma aplicação *mobile*, onde, após a autenticação do usuário através do *Amazon Cognito*, será possível realizar requisições HTTP ao *API Gateway*. Este, irá manipular as requisições a fim de disparar a função *Lambda* atrelada à rota especificada na requisição. A função *Lambda*, por sua vez, receberá os parâmetros enviados via HTTP e executará uma lógica a fim de realizar operações de escrita e/ou leitura em tabelas do *DynamoDB*. Após realizar a respectiva tarefa, a função *Lambda* retorna o resultado da operação ao *API Gateway*, que devolve a informação à interface de usuário.

#### 4.1.1 Autenticação

Ao se cadastrar na aplicação, é feita a inclusão de um usuário em um *user pool*, essa inclusão fica aguardando uma confirmação da parte do usuário. O cadastro é realizado através de um endereço de *e-mail* válido, portanto, essa informação deve ser única para cada usuário.



**Figura 4** – Arquitetura simplificada do sistema

Ao realizar o cadastro é enviado ao *e-mail* informado um código de ativação, que deve ser recuperado pelo usuário e enviado de volta via aplicativo ao *user pool* do *Amazon Cognito* a fim de validar o cadastro do usuário.

Uma vez com o cadastro realizado e validado, o usuário poderá efetuar *login* na aplicação. Ao entrar, o usuário receberá um *token* de autenticação que o associa a um *identity pool* que o concede acesso temporário a recursos da aplicação na nuvem. Para este caso, o único recurso que o usuário possui acesso com o *token* de autenticação são os serviços *API Gateway* e *Amazon Cognito*.

#### 4.1.2 Comunicação com a API

Uma vez autenticado, o usuário pode realizar requisições **HTTP** ao *API Gateway*. As chamadas de **API** são realizadas à medida que o usuário interage com a aplicação. Cada chamada de **API** será recebida e manipulada pelo *API Gateway* que, por sua vez, irá disparar o microserviço correspondente à rota informada na requisição **HTTP**. Uma lista com as chamadas de **API** para cada microserviço, suas rotas, funções *Lambda* relacionadas e tabelas *DynamoDB* alteradas é mostrada no Apêndice A.

Ao *API Gateway* é concedida uma *role* do **IAM**, assim, ao receber uma requisição HTTP e identificar qual o microserviço associado à rota daquela requisição, o *API Gateway* é capaz de invocar a função *Lambda* correspondente. Para esse caso, foi concedido ao *API Gateway* acesso total às funções *Lambda* do *backend*, a fim de que seja possível invocá-las e receber suas respostas.

#### 4.1.3 Processamento

Ao ser acionada pelo *API Gateway*, a função *Lambda* recebe os parâmetros enviados na requisição e que serão utilizados para realização das operações nos bancos de dados. Em cada função é executado um código que inicializa uma instância da classe *AWS.DynamoDB.DocumentClient*, importada do pacote *AWS-SDK* e realiza operações de inserção, alteração, deleção e consulta às tabelas do *DynamoDB*.

#### 4.1.4 Armazenamento

O armazenamento foi feito com o *DynamoDB*, foram utilizadas quatro tabelas, as quais são mostradas e detalhadas no Apêndice B. Os esquemas de chaves e atributos foram escolhidos pensando nas funcionalidades da aplicação. Foram adotadas duas estratégias de relacionamentos um para muitos, uma discussão dos pros e contras de cada uma no *DynamoDB* será apresentada no capítulo seguinte.

## 4.2 DESENVOLVIMENTO DO BACKEND

Uma vez definidas as tecnologias e métodos a serem utilizados, iniciou-se a implementação do *backend* da aplicação na nuvem. A seguir são listadas as fases deste processo.

### 4.2.1 Usuário do IAM

O processo de criação de um usuário IAM é bastante simples e intuitivo, pode ser feito integralmente através do console da AWS, que é a plataforma online para gerenciamento de recursos e serviços AWS. Basta acessar o console com a conta AWS, procurar pelo serviço IAM e adicionar um novo usuário. Ao iniciar o processo de criação será possível escolher entre o tipo de acesso que o usuário possuirá: pragmático ou acesso ao console de gerenciamento AWS, como é visto na Figura 5.

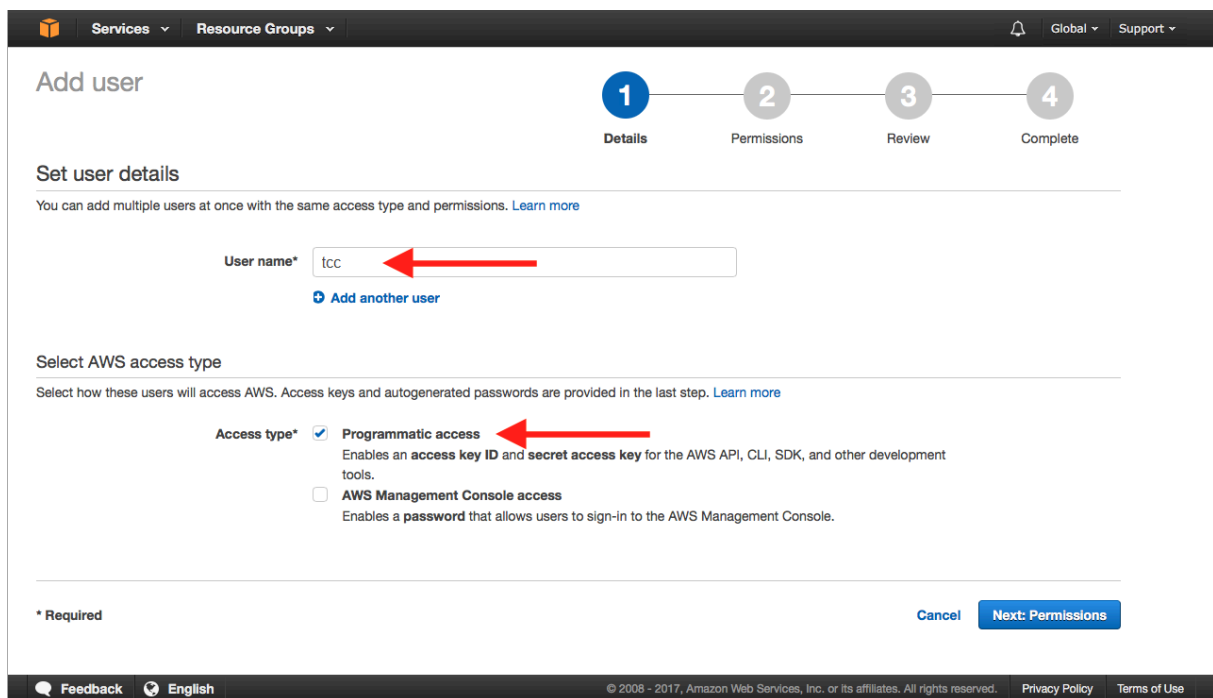
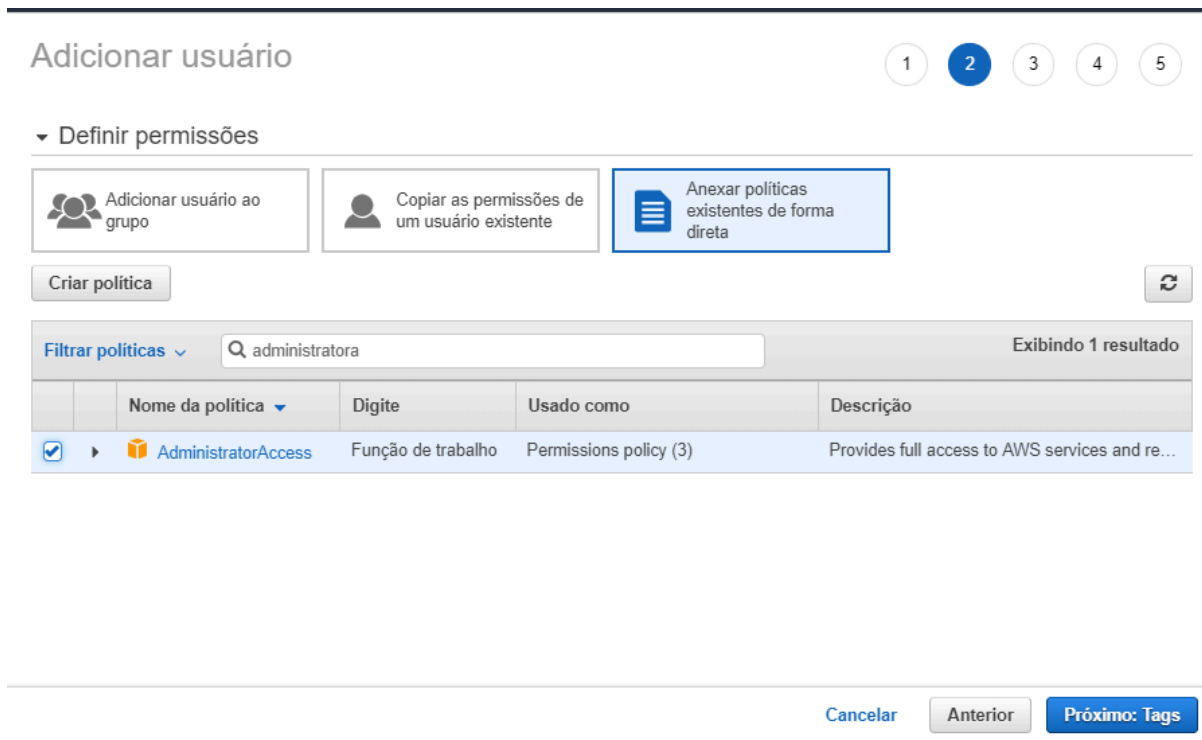
The image is a screenshot of the AWS IAM 'Add user' console. At the top, there's a navigation bar with 'Services' and 'Resource Groups' dropdowns, and a top right corner with a notification bell, 'Global' region, and 'Support' link. Below the navigation bar, the 'Add user' page is shown. It has a progress indicator with four steps: 1. Details (active), 2. Permissions, 3. Review, and 4. Complete. The 'Set user details' section is the main focus. It includes a sub-header 'Set user details' and a note: 'You can add multiple users at once with the same access type and permissions. [Learn more](#)'. There is a 'User name\*' text input field containing 'tcc', with a red arrow pointing to it. Below the input field is a link '+ Add another user'. The 'Select AWS access type' section follows, with a sub-header and a note: 'Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)'. Under 'Access type\*', there are two radio button options. The first is 'Programmatic access', which is selected (checked), and has a red arrow pointing to it. Its description is 'Enables an access key ID and secret access key for the AWS API, CLI, SDK, and other development tools.' The second option is 'AWS Management Console access', which is not selected. Its description is 'Enables a password that allows users to sign-in to the AWS Management Console.' At the bottom of the form, there is a '\* Required' label, a 'Cancel' button, and a 'Next: Permissions' button. The footer of the console shows 'Feedback', 'English' language selector, and copyright information: '© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.' along with links for 'Privacy Policy' and 'Terms of Use'.

Figura 5 – Criação do usuário IAM e tipos de acesso do usuário IAM

- **Acesso pragmático** é um tipo de acesso que gera uma chave ID de acesso (*access key ID*) e uma chave de acesso secreta (*access secret key*), as quais permitem a integração do usuário IAM a ferramentas de desenvolvimento de aplicações, tais como AWS-SDK, AWS *Amplify* dentre outras.
- **Acesso ao console de gerenciamento AWS** fornece uma senha para que os operadores consigam acessar o console da AWS e gerenciar os recursos da nuvem para manutenção da aplicação desenvolvida.

Como a aplicação ainda está sendo desenvolvida, o acesso escolhido foi o pragmático. Em seguida, é necessário anexar uma política de permissões ao usuário IAM, a política de

acesso irá definir quais acessos esse usuário possuirá. Conforme a Figura 6 será anexada a política de administrador, que concederá ao usuário do IAM total acesso aos recursos de nuvem da aplicação.



**Figura 6** – Anexação de políticas de permissão ao usuário IAM

#### 4.2.1.1 Chaves de acesso

Em seguida é feita uma revisão do usuário que será criado, e, ao confirmar a criação, será disponibilizada a chave ID de acesso e a chave de acesso secreta. Essas chaves devem ser armazenadas em um local específico para que sejam usadas no processo de criação e *deploy* das aplicações a serem desenvolvidas. Para essa configuração foi utilizada a ferramenta AWS CLI, uma interface de linha de comando oferecida pela AWS para gerenciamento de aplicações baseadas na nuvem da AWS. Via prompt de comando, foi executado o comando:

```
1 $ aws configure
```

Após seguir o passo a passo mostrado no fluxo do comando e inserir as chaves geradas na criação do usuário IAM as chaves de acesso estão configuradas e passarão a ser automaticamente utilizadas em aplicações a serem desenvolvidas.

#### 4.2.2 User pool

O próximo passo foi a criação de um *User pool*. O processo também é feito através do console AWS, na página do serviço *Amazon Cognito*. Para criar um *user pool* é necessário nomeá-lo e em seguida é possível atribuir uma série de propriedades, tais como atributos, tipos



de verificação de conta, políticas de privacidade dentre outras. Para o *user pool* da aplicação foi determinado que seria utilizado o atributo *e-mail* como propriedade de *login* e verificação.

Após concluir o processo de criação, são fornecidas duas informações importantes: o *pool id* e o ARN do *user pool*. A primeira propriedade é um identificador do recurso localmente (dentro da conta AWS), já a segunda é um identificador global do recurso, a sigla ARN significa *AWS Resource Name* (Nome de Recurso AWS). Essas informações, bem como a região em que o *user pool* foi criado serão necessárias posteriormente.

#### 4.2.2.1 Adicionar uma aplicação cliente

Após finalizado o processo de criação do *user pool* será adicionada uma aplicação como cliente do recurso criado. Dessa forma é gerada uma chave de aplicação cliente (*App client ID*) que posteriormente será configurada na aplicação na nuvem, a fim de atrelar a aplicação ao *user pool* criado.

#### 4.2.3 Tabelas do *DynamoDB*

A criação de tabelas no *DynamoDB* também é simples, intuitivo e pode ser realizado através do console da AWS. Após definidas as tabelas e seus esquemas de chaves basta criar a tabela na página do serviço *DynamoDB* conforme a Figura 7.

**Criar tabela do DynamoDB** Tutorial ?

O DynamoDB é um banco de dados sem esquema que requer somente o nome de uma tabela e a chave primária. A chave primária da tabela é constituída de um ou dois atributos que identificam itens, particionam os dados, e classificam os dados dentro da partição de maneira exclusiva.

Nome da tabela\*  ⓘ

Chave primária\* Chave de partição

String ⓘ

☒ Adicionar chave de classificação

String ⓘ

**Configurações da tabela**

As configurações padrão são a forma mais rápida de começar a usar sua tabela. Você poderá modificar essas configurações padrão agora ou depois que a tabela for criada.

☒ Usar configurações padrão

- Nenhum índice secundário.
- Capacidade provisionada definida para 5 leituras e 5 gravações.
- Alarmes básicos com 80% de limite superior usando o tópico do SNS "dynamodb".
- Encryption at Rest with DEFAULT encryption type.

+ Add Tags **NOVIDADE!**

Cobranças adicionais podem ser aplicadas se você exceder os níveis do CloudWatch ou Simple Notification Service do nível gratuito da AWS. As configurações avançadas de alarme estão disponíveis no console de gerenciamento do CloudWatch.

Cancelar Criar

**Figura 7** – Criação da tabela eventos

#### 4.2.4 Codificação das APIs

Com as configurações anteriores concluídas foi iniciada a codificação das APIs com o *Serverless Framework*. Ao instalar a ferramenta com um único comando é criado um novo projeto *Node.js* baseado no ambiente de nuvem da AWS.

---

```
1 $ serverless create --template aws-nodejs --path \textit{mobile} --serverless-app
```

---

Após a criação do projeto, são gerados dois arquivos importantes: *handler.js* e *serverless.yml*.

- O *handler.js* é o arquivo que contém os códigos das funções que, de fato, serão publicadas como funções *Lambda*.
- O *serverless.yml* possui a descrição da estrutura e configuração dos serviços AWS que serão implantados.

Para cada função criada é necessário configurar seu *endpoint* de API no arquivo *serverless.yml*. Um trecho do arquivo *serverless.yml* contendo informações de configuração de ambiente e de uma função *Lambda* é disponibilizado no Apêndice D.

Uma vez com o ambiente construído e configurado, para implantar o *backend* construído basta executar o comando a seguir:

---

```
1 $ serverless deploy
```

---

#### 4.2.5 Identity Pool

Para finalizar o *backend* da aplicação é necessário criar um *identity pool*, que, como visto anteriormente, irá especificar a quais recursos os usuários terão acesso ao efetuar *login* na aplicação. O processo de criação do *identity pool* pode ser realizado através do console da AWS na página do *Amazon Cognito*. Para criá-lo é necessário especificar um nome e os provedores de autenticação suportados, nesse caso o provedor será o próprio *Amazon Cognito*. Ao utilizar o *Amazon Cognito* como provedor de autenticação será necessário especificar o *user pool* da aplicação informando o *pool id* e o ARN do *user pool* gerados na criação do *user pool*.

Por fim, é necessário associar duas *roles* do IAM à identidade federada criada, uma para usuários autenticados e outra para usuários não autenticados. Para os usuários não autenticados foi definida uma *role* que não concede nenhuma permissão de acesso os recursos da aplicação, a fim de que, para realizar qualquer ação, o usuário precise estar autenticado via *Amazon Cognito*. Já para os usuários autenticados, a fim de agilizar o desenvolvimento, foi definida uma *role* padrão que concede acesso total aos recursos da aplicação.

Ao finalizar a criação do *identity pool* é disponibilizado um identificador para o recurso, chamado *identity pool id*, que também será utilizado na integração com o *frontend*.

## 4.3 INTERFACE DE USUÁRIO

Para finalizar o sistema foi implementada uma aplicação *mobile* para interação com o ambiente na nuvem. A interface da aplicação foi dividida em dez telas que se encontram listadas no Apêndice C, juntamente com as informações de quais APIs populam cada tela quais serviços são chamados em cada tela e uma breve descrição. A seguir são listadas as etapas de criação da interface e sua integração com o ambiente na nuvem.

### 4.3.1 AWS Amplify

A fim de permitir que a aplicação *Mobile* consiga se comunicar com o *backend*, é necessário incluir no projeto *React Native* a biblioteca *AWS Amplify* e configurá-la.

A configuração é feita no arquivo *config.js* na pasta raiz do projeto *mobile* conforme mostrado no Trecho de código fonte [4.1](#)

#### Trecho de Código 4.1 – Conteúdo do arquivo de configuração *config.js*

```
1  export default {
2    apiGateway: {
3      REGION: "us-east-1",
4      URL: "APIGATEWAYURL"
5    },
6    cognito: {
7      REGION: "us-east-1",
8      USER_POOL_ID: "COGNITOUSERPOOLID",
9      APP_CLIENT_ID: "COGNITOAPPCLIENTID",
10     IDENTITY_POOL_ID: "IDENTITYPOOLID"
11   }
12 };
```

Analisando o arquivo *config.js* fica evidente que os únicos Serviços aos quais o aplicativo *Mobile* possui acesso são o *Amazon Cognito* e o *API Gateway*. O valor *APIGATEWAYURL* é a URL do *API Gateway*, essa URL é gerada ao se implantar o *backend* e pode ser consultada através da página do *API Gateway* no console AWS. Já os valores *COGNITOUSERPOOLID*, *COGNITOAPPCLIENTID* e *IDENTITYPOOLID* são, respectivamente, o *pool id*, *App client ID* e *identity pool id*, todos citados anteriormente.

Com a configuração realizada é adicionado um trecho de código no ponto de entrada da aplicação para iniciar a conexão entre *frontend* e *backend*.

O código de configuração mostrado no Trecho de código [4.2](#) evidencia que o cadastro na aplicação é obrigatório através da propriedade *mandatorySignIn*. O valor *APINAME* é o nome do serviço *API Gateway*, conforme especificado no arquivo *serverless.yml*.

---

**Trecho de Código 4.2** – ConFiguração da conexão *frontend-backend* no *entrypoint* da aplicação *Mobile*

---

```
13 import Amplify from "aws-amplify";
14 import config from "../config";
15
16 Amplify.configure({
17   Auth: {
18     mandatorySignIn: true,
19     region: config.cognito.REGION,
20     userPoolId: config.cognito.USER_POOL_ID,
21     identityPoolId: config.cognito.IDENTITY_POOL_ID,
22     userPoolWebClientId: config.cognito.APP_CLIENT_ID
23   },
24   API: {
25     endpoints: [
26       {
27         name: "APINAME",
28         endpoint: config.apiGateway.URL,
29         region: config.apiGateway.REGION
30       },
31     ]
32   }
33 });
```

---

## 5 RESULTADOS

Finalmente foi possível criar e implantar a aplicação conforme especificado. Esse capítulo tratará sobre possíveis dificuldades na implementação do projeto e os resultados obtidos. Imagens mostrando a aplicação funcionando são apresentadas no Apêndice E.

### 5.1 API GATEWAY

Não houve grandes complicações para configuração do serviço *API Gateway*, uma vez que o processo é integralmente automatizado pelo *Serverless Framework* desde que o arquivo de configuração *serverless.yml* tenha sido criado corretamente descrevendo todos os microserviços e suas rotas. Ainda assim, a fim de explorar os resultados, é possível acessar a página do *API Gateway* no console AWS, logado com a conta AWS, e realizar testes de requisições HTTP para a API, conforme apresentado na Figura 8.

Solicitação: `/get-usuario/igorgonribsilva@gmail.com`

Status: 200

Latência: 562 ms

Corpo da resposta

```
{
  "result": {
    "statusCode": 200,
    "headers": {
      "Access-Control-Allow-Origin": "*",
      "Access-Control-Allow-Credentials": true
    },
    "body": "{\\"contatos\\": [\\"igorgon1@hotmail.com\\"], \\"username\\": \\"igorgonribsilva@gmail.com\\", \\"agenda\\": [ { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"javnvkjsdlnvkjn\\", \\"idEvento\\": \\"f1d76ddb-6e9c-4876-b057-25c6d9488ae0\\", { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"aehocaraio\\", \\"idEvento\\": \\"9aableb2-7b7f-4b37-9710-b081a2343a81\\", { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"testetesteset\\", \\"idEvento\\": \\"7476fef0-9309-4800-b26f-bc26f88f835f\\", { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"teste datepicke r\\", \\"idEvento\\": \\"d267cb83-bf8f-45e9-92be-f8ca50d1e88c\\", { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"sda fdsfasfa\\", \\"idEvento\\": \\"58a0638b-a457-4755-9a6c-43735f5f8cf4\\", { \\"criador\\": \\"igorgonribsilva@gmail.com\\", \\"nomeEvento\\": \\"teste checkbox\\", \\"idEvento\\": \\"c58284e9-fce2-4db3-b2a5-09f90e551fc2\\"} } } } } } }
```

**Figura 8** – Teste de método *get-usuario* via console AWS

Entretanto, um resultado diferente é obtido ao tentar realizar a mesma chamada através

do *software Postman*. Nesse caso a requisição é feita sem as informações de autenticação. Como resultado, é retornada uma simples mensagem com o texto "*Missing Authentication Token*" conforme a Figura 9. Esse teste comprova que a API só é acessada por usuários autenticados.

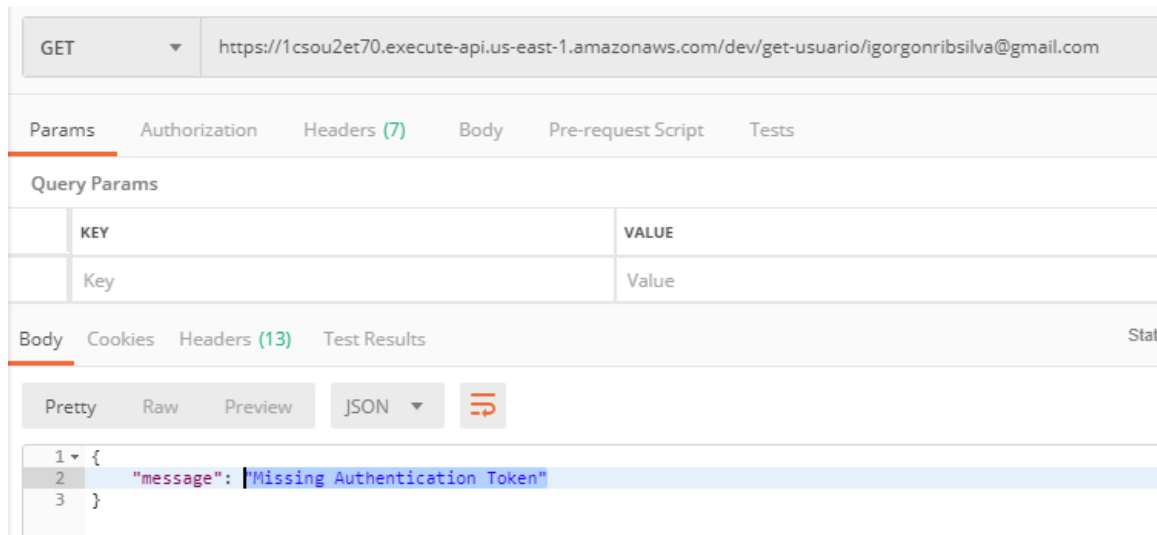


Figura 9 – Teste de método *get-usuario* via *Postman*

## 5.2 LAMBDA

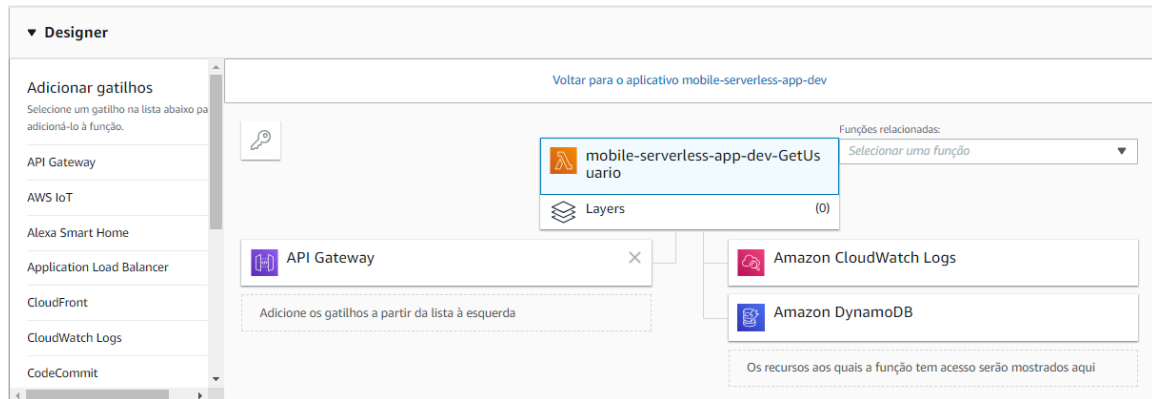
No desenvolvimento das funções *Lambda* também não houve grandes dificuldades, já que o *deploy* também é automático e, principalmente, devido ao suporte de várias tecnologias e linguagens de programação oferecido pela AWS, permitindo que o desenvolvedor trabalhe com as tecnologias com as quais possui afinidade.

Através do console AWS é possível visualizar algumas informações sobre as funções implantadas. Uma delas é o esquema de acessos de cada função, conforme Figura 10. À esquerda são listados os serviços que podem disparar a função e à direita os serviços que podem ser acessados pela função. Também é possível adicionar gatilhos e acessos da função através do console.

As funções podem ser disparadas apenas pelo *API Gateway* e podem acessar o *DynamoDB* para realizar as transações com o banco de dados, e o *Amazon Cloud Watch* para registrar logs de execução a fim de auxiliar o desenvolvimento e a manutenção. Através do console também é possível monitorar as execuções da função *Lambda* e modificar seu código.

## 5.3 DYNAMODB

Apesar dos resultados satisfatórios em termos de latência, o modelo de banco de dados proposto trouxe uma série de complicações em termos de relacionamento. Algumas das técnicas de associação de entidades adotadas foram semelhantes às dos bancos de dados relacionais,



**Figura 10** – Esquema de acessos da função *getUsuario* visualizado através do console AWS

e, uma vez que o relacionamento em bancos de dados NoSQL não é consistente como nos bancos de dados tradicionais, cabe ao desenvolvedor implementar uma lógica que garanta a consistência e confiabilidade da melhor forma possível.

Bancos de dados não relacionais costumam ser minimalistas em termos de número de tabelas. No modelo proposto foram adotadas técnicas de aninhamento de algumas propriedades para reduzir o número de tabelas. Porém, para relacionamentos de muitos para muitos foram criadas tabelas intermediárias. Essa estratégia elevou a complexidade do sistema além do necessário, complexidade esta que poderia ser reduzida adotando-se um modelo mais simples, com apenas duas, ou até mesmo uma única tabela, o que implicaria também em um menor número de microserviços.

Outra possível estratégia para reduzir o número de tabelas, e, consequentemente, a complexidade do sistema seria a utilização de índices secundários, um recurso oferecido pela AWS onde os dados da tabela são duplicados com esquemas de chaves de partição e classificação diferentes da tabela original, dando assim, mais flexibilidade para as consultas sem a criação de novas entidades. Apesar de ser cogitado o uso desse recurso no modelo adotado, foi decidido não utilizar índices secundários devido aos custos financeiros, os quais seriam incrementados para manter os índices, já que os mesmos são, na verdade, duplicação de dados, e, portanto, exigem mais recursos de *hardware* para serem mantidos.

## 6 CONCLUSÃO

A computação em nuvem vem oferecendo a desenvolvedores de diversos lugares e níveis de experiência novas soluções para problemas antigos, deve-se, portanto apresentar as vantagens e desvantagens dessas novas opções.

Em termos gerais a aplicação atingiu os objetivos almejados, apresentando uma experiência de usuário segura e consistente e um tempo de resposta reduzido e satisfatório.

### 6.0.1 Vantagens

Cabe ressaltar como grandes vantagens dos métodos utilizados a auto escalabilidade dos serviços de processamento e armazenamento utilizados, o que tira das mãos do desenvolvedor a preocupação com recursos de infraestrutura. Outro ponto de vantagem é o sistema de cobrança *pay per use*, que reduz os custos de processamento, levando o desenvolvedor a pagar apenas pelo tempo em que o serviço foi executado.

Outra vantagem de sistemas implantados em nuvem é a facilidade de configuração, implantação e manutenção oferecida através de *frameworks* especializados no assunto bem como através da plataforma da empresa provedora dos serviços de nuvem. Até mesmo uma pessoa com pouco conhecimento em computação poderia ser treinada para monitorar uma aplicação através do console AWS graças às facilidades que este oferece aos seus clientes.

### 6.0.2 Desvantagens

Entretanto, como principal desvantagem em desenvolver um sistema utilizando os serviços aqui utilizados está a grande diferença de análise na elaboração de um modelo de dados para armazenamento de informações. Ao modelar um banco de dados para o *DynamoDB* deve-se partir da suposição de que já são conhecidas as funcionalidades da aplicação e estabelecer tabelas de forma a alimentar o *frontend* conforme as funcionalidades (AWS, 2019d). Essa análise pode parecer estranha a uma pessoa acostumada a trabalhar com bancos de dados relacionais e implicar em uma longa curva de aprendizado, ou até mesmo, drásticas mudanças de projeto após já iniciado o desenvolvimento da aplicação.

## 6.1 TRABALHOS FUTUROS

Durante o desenvolvimento do projeto e da escrita deste texto foram encontrados pontos que poderiam ser tratados para melhoria da aplicação e da experiência do usuário. Esses pontos são listados abaixo.



- No *identity pool*, ao especificar os acessos que os usuários federados possuiriam ao se autenticarem na aplicação foi definido que possuiriam, temporariamente, acesso total aos recursos de nuvem da aplicação. Essa não é uma boa prática, o ideal seria especificar cada um dos recursos os usuários poderiam acessar, bem como os seus níveis de acesso a cada recurso, ou seja, quais ações o usuário poderia executar em cada recurso. Assim, seria interessante especificar permissão para chamar métodos do *API Gateway* e algumas permissões do *Amazon Cognito*, tais como alteração de senha, por exemplo.
- Outro ponto a ser melhorado no futuro é realizar uma remodelagem do banco de dados para uma estrutura mais simples. Bancos de dados bem estruturados que utilizam o *DynamoDB* comumente possuem apenas uma tabela (AWS, 2019d). Ao remodelar o banco de dados com o objetivo de simplificá-lo estaremos tornando as transações com o *DynamoDB* mais seguras, confiáveis e mais fáceis de serem implementadas.
- Uma alternativa ao banco de dados não relacional seria o uso de um banco de dados relacional escalável. A AWS oferece o serviço *Relational Database Service* (RDS) que facilita o gerenciamento de bancos de dados relacionais possibilitando implantá-los com alta escalabilidade e disponibilidade. O RDS é compatível com diversos bancos de dados, tais como *Oracle*, *Microsoft SQL Server*, *MySQL* e *Postgres* e oferece recursos que auxiliam na migração de bancos já existentes para a nuvem.
- Uma vez remodelado o banco de dados seria necessário realizar uma adaptação das funções para o atendê-lo. Um número menor de funções *Lambda* seria necessário, o que implicaria em menores custos para manter o sistema.
- Também seria interessante a utilização do serviço *Simple Notification Service* (SNS), um serviço da AWS que envia notificações quando ocorrem eventos em alguns serviços AWS. Dessa forma pode-se disparar uma notificação ao realizar alguma ação em tempo real. Isto pode ser interessante para manter a consistência entre os registros do banco de dados, pois seria possível, por exemplo, disparar uma função *Lambda* que insere convidados e convites sempre que um novo evento fosse adicionado, ou ainda deletar as informações relacionadas a um evento nas demais tabelas assim que ocorra uma deleção na tabela de eventos.
- A fim de melhorar ainda mais a experiência do usuário, poderia-se incluir um sistema de envio de *e-mails* para os usuário cadastrados sempre que o usuário em questão fosse convidado a um novo evento. A AWS possui um serviço de envio de *e-mails*, denominado *Simple E-mail Service* (SES), que pode facilmente ser acionado quando um evento ocorrer.

## REFERÊNCIAS

ABRAMOVA, Veronika; BERNARDINO, Jorge. Nosql databases: Mongodb vs cassandra. In: ACM. **Proceedings of the international C\* conference on computer science and software engineering**. [S.l.], 2013. p. 14–22. Citado na página 21.

AWS. **Amazon API Gateway Guia do desenvolvedor**. 2019. Disponível em: <[https://docs.aws.amazon.com/pt\\_br/apigateway/latest/developerguide/apigateway-dg.pdf#api-ref](https://docs.aws.amazon.com/pt_br/apigateway/latest/developerguide/apigateway-dg.pdf#api-ref)>. Acesso em: 2019-06-02. Citado na página 22.

\_\_\_\_\_. **Amazon CloudWatch**. 2019. Disponível em: <<https://aws.amazon.com/pt/cloudwatch/>>. Acesso em: 2019-06-02. Citado na página 24.

\_\_\_\_\_. **Amazon Cognito Guia do desenvolvedor**. 2019. Disponível em: <[https://docs.aws.amazon.com/pt\\_br/cognito/latest/developerguide/cognito-dg.pdf#what-is-amazon-cognito](https://docs.aws.amazon.com/pt_br/cognito/latest/developerguide/cognito-dg.pdf#what-is-amazon-cognito)>. Acesso em: 2019-06-02. Citado na página 23.

\_\_\_\_\_. **Amazon DynamoDB: API Reference**. 2019. Disponível em: <[https://docs.aws.amazon.com/pt\\_br/amazondynamodb/latest/developerguide/dynamodb-dg.pdf#Introduction](https://docs.aws.amazon.com/pt_br/amazondynamodb/latest/developerguide/dynamodb-dg.pdf#Introduction)>. Acesso em: 2019-06-02. Citado 6 vezes nas páginas 8, 24, 25, 26, 39 e 40.

\_\_\_\_\_. **Amazon DynamoDB Guia do desenvolvedor Versão da API 2012-08-10**. 2019. Disponível em: <[https://docs.aws.amazon.com/pt\\_br/cognito/latest/developerguide/what-is-amazon-cognito.html](https://docs.aws.amazon.com/pt_br/cognito/latest/developerguide/what-is-amazon-cognito.html)>. Acesso em: 2019-06-02. Citado na página 24.

\_\_\_\_\_. **AWS Amplify Console User Guide**. 2019. Disponível em: <<https://docs.aws.amazon.com/amplify/latest/userguide/amplify-console-ug.pdf>>. Acesso em: 2019-06-02. Citado na página 26.

\_\_\_\_\_. **AWS Identity and Access Management API Reference API Version 2010-05-08**. 2019. Disponível em: <<https://docs.aws.amazon.com/IAM/latest/APIReference/iam-api.pdf#Welcome>>. Acesso em: 2019-06-02. Citado na página 24.

\_\_\_\_\_. **AWS Lambda Guia do desenvolvedor**. 2019. Disponível em: <[https://docs.aws.amazon.com/pt\\_br/lambda/latest/dg/lambda-dg.pdf#API\\_Reference](https://docs.aws.amazon.com/pt_br/lambda/latest/dg/lambda-dg.pdf#API_Reference)>. Acesso em: 2019-06-02. Citado na página 23.

\_\_\_\_\_. **Computação em nuvem com a AWS**. 2019. Disponível em: <<https://aws.amazon.com/pt/what-is-aws/>>. Acesso em: 2019-06-02. Citado na página 22.

\_\_\_\_\_. **Microsserviços**. 2019. Disponível em: <<https://aws.amazon.com/pt/microservices/>>. Acesso em: 2019-06-02. Citado 2 vezes nas páginas 19 e 20.

\_\_\_\_\_. **O que é o docker?** 2019. Disponível em: <[https://aws.amazon.com/pt/docker/?nc1=f\\_cc](https://aws.amazon.com/pt/docker/?nc1=f_cc)>. Acesso em: 2019-06-11. Citado na página 18.

\_\_\_\_\_. **SaaS na AWS Maximize a inovação e a agilidade criando uma solução de SaaS na AWS**. 2019. Disponível em: <<https://aws.amazon.com/pt/partners/saas-on-aws/>>. Acesso em: 2019-06-11. Citado na página 18.

\_\_\_\_\_. **Tipos de computação em nuvem**. 2019. Disponível em: <<https://aws.amazon.com/pt/types-of-cloud-computing/>>. Acesso em: 2019-06-02. Citado na página 17.

AZURE, Microsoft. **O que é IaaS? Infraestrutura como serviço**. 2019. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-iaas/>. Acesso em: 2019-06-11. Citado na página 17.

\_\_\_\_\_. **O que é PaaS? Plataforma como serviço**. 2019. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-paas/>. Acesso em: 2019-06-11. Citado na página 17.

\_\_\_\_\_. **O que é SaaS? Software como serviço**. 2019. Disponível em: <https://azure.microsoft.com/pt-br/overview/what-is-saas/>. Acesso em: 2019-06-11. Citado 3 vezes nas páginas 8, 16 e 18.

CATTELL, Rick. Scalable sql and nosql data stores. **Acm Sigmod Record**, ACM, v. 39, n. 4, p. 12–27, 2011. Citado na página 21.

FACEBOOK. **React Native Build native mobile apps using JavaScript and React**. 2017. Disponível em: <https://facebook.github.io/react-native/>. Acesso em: 2019-06-02. Citado na página 27.

FOWLER, Martin; LEWIS, James. Microservices a definition of this new architectural term. **URL: http://martinfowler.com/articles/microservices.html**, 2014. Citado 4 vezes nas páginas 8, 18, 19 e 20.

JUNIOR, I. S. MENEZES et al. Análise comparativa entre um banco de dados relacional (postgresql) e um banco de dados não somente sql (mongodb). Universidade de Pernambuco, 2018. Citado na página 21.

PIRES, Jackson. **O que é API? REST e RESTful? Conheça as definições e diferenças!** 2017. Disponível em: <https://becode.com.br/o-que-e-api-rest-e-restful/>. Acesso em: 2019-06-02. Citado na página 22.

SERVERLESS. **Why Serverless?** 2018. Disponível em: <https://serverless.com/learn/overview/>. Acesso em: 2019-06-02. Citado na página 26.

TAMANE, Sharvari. Non-relational databases in big data. In: ACM. **Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies**. [S.l.], 2016. p. 134. Citado na página 21.

## **APÊNDICES**

## APÊNDICE A – RELAÇÃO DE MICROSERVIÇOS DO BACKEND

**Tabela 1** – Microserviços do sistema.

Função	Rota	Método	Descrição
<i>putUsuario</i>	<i>put-Usuario</i>	POST	Insere/Atualiza um usuário na tabela <i>usuarios</i>
<i>getEvento</i>	<i>get-Evento</i>	GET	Busca um evento na tabela <i>eventos</i> através da chave de partição <i>idEvento</i>
<i>putEvento</i>	<i>put-Evento</i>	POST	Insere/Atualiza um evento na tabela <i>eventos</i>
<i>listEvento</i>	<i>list-Evento</i>	GET	Busca uma lista de eventos na tabela <i>eventos</i> através da chave de partição <i>criador</i>
<i>listMeusConvites</i>	<i>list-meus-convites</i>	GET	Busca os convites enviados ao usuário na tabela <i>meus-convites</i> através da chave de classificação <i>username-convidado</i>
<i>putMeusConvites</i>	<i>put-meus-convites</i>	PUT	Insere/Atualiza um convite na tabela <i>meus-convites</i>
<i>respostaConvite</i>	<i>resposta-convite</i>	PUT	Envia confirmação ou rejeição de convite a um evento inserindo ou alterando a tabela <i>convidados-evento</i> e deletando na tabela <i>meus-convites</i>
<i>listConvidados</i>	<i>list-convidados</i>	GET	Busca os convidados a um evento na tabela <i>convidados-evento</i> através da chave de classificação
<i>putConvidado</i>	<i>put-Convidado</i>	PUT	Insere/Atualiza na tabela <i>convidados-evento</i> e na tabela <i>meus-convites</i>
<i>getUsuario</i>	<i>get-Usuario</i>	GET	Busca um usuário na tabela <i>usuarios</i> através da chave de partição

## APÊNDICE B – RELAÇÃO DE TABELAS NO DYNAMODB

**Tabela 2** – Tabelas *DynamoDB*.

Tabela	Chave de Partição	Chave de Classificação	Descrição
usuarios	<i>username</i>		Armazena os usuários no sistema, o valor da chave de partição <i>username</i> é o <i>e-mail</i> utilizado para cadastro. Também são armazenados nesta tabela os atributos <i>agenda</i> (uma lista de eventos e <i>contatos</i> (uma lista de <i>usernames</i> de usuários))
eventos	criador	<i>idEvento</i>	Armazena os eventos. A chave de partição <i>idEvento</i> é utilizada para recuperar um evento individual e a chave de classificação para recuperar os eventos criados por um usuário. Também são armazenados nesta tabela a data, local, descrição e nome do evento.
meus-convites	<i>usernameConvidado</i>	<i>idEvento</i>	Armazena a relação de convites enviados a cada usuário. A combinação das chaves de classificação e partição permite recuperar os eventos para os quais um usuário foi convidado para popular a tela <i>Convites</i> do frontend. Outras informações contidas nesta tabela são: o usuário que enviou o convite (atributo <i>de</i> ) e o nome do evento (atributo <i>nomeEvento</i> ).
convidados-evento	<i>idEvento</i>	<i>username</i>	Armazena a relação de quais usuários foram convidados a cada evento. Sua combinação de chaves permite listar os usuários convidados a cada evento a fim de que o usuário que criou o evento tenha acesso à relação de convidados que ainda não responderam ao convite, os que aceitaram e os que negaram. Uma outra informação armazenada nesta tabela é o atributo <i>confirma</i> , que é a resposta ao convite. Se o atributo não estiver presente para um registro significa que o convidado ainda não respondeu.

## APÊNDICE C – RELAÇÃO DE TELAS DA INTERFACE DE USUÁRIO

**Tabela 3** – Telas da Interface de usuário.

Tela	Serviço que popula a tela	Serviços chamados	Descrição
Cadastro	Populada pelo usuário	Serviços de <i>sign up</i> e de confirmação de cadastro do <i>Amazon Cognito</i>	Tela para realizar cadastro de usuário via <i>e-mail</i> e confirmação do cadastro através de um código de verificação enviado para o <i>e-mail</i> informado.
Login	Populada pelo usuário	Serviço de autenticação do <i>Amazon Cognito</i>	Realiza <i>login</i> na aplicação com <i>e-mail</i> e senha do usuário cujo cadastrado esteja finalizado.
Agenda	Microserviço <i>getUsuario</i>	API <i>getEvento</i>	Lista os eventos na agenda do usuário autenticado. Estes eventos são armazenados no atributo <i>agenda</i> da tabela <i>usuarios</i> de forma aninhada. Ao clicar em um item da lista é chamada a API <i>getEvento</i> para buscar as informações do evento selecionado e invocar a tela <i>Visualizar Evento</i> .
Contatos	Microserviço <i>getUsuario</i>		Lista os contatos do usuário autenticado. Estes contatos são armazenados no atributo <i>contatos</i> da tabela <i>usuarios</i> de forma aninhada.
Eventos	Microserviço <i>Visualizar Evento</i>	Microserviço <i>getEvento</i>	Lista os eventos criados pelo usuário autenticado. Estes contatos são armazenados na tabela <i>eventos</i> . Ao clicar em um item da lista é invocada a tela <i>Visualizar Evento</i> .
Visualizar evento	Microserviço <i>getEvento</i> e Microserviço <i>listConvidados</i>		Mostra os detalhes do evento selecionado eventos criados pelo usuário autenticado e os usuários que foram convidados ao evento, bem como a resposta de cada um ao convite.
Convites	Microserviço <i>listMeusConvites</i>	Microserviço <i>getEvento</i>	Lista os convites enviados ao usuário autenticado e que ainda não foram respondidos. Ao clicar em um item da lista é chamada a tela <i>Visualizar convite</i> .
Visualizar convite	Microserviço <i>getEvento</i>	Microserviço <i>respostaConvite</i>	Mostra os detalhes do evento para o qual o usuário foi convidado. É possível responder ao convite, aceitando ou recusando.
Adicionar Evento	Populada pelo usuário	Microserviço <i>putEvento</i>	Inclui um evento na tabela <i>eventos</i> .
Adicionar Contato	Populada pelo usuário	Microserviço <i>putUsuario</i>	Inclui um contato no atributo <i>contatos</i> da tabela <i>usuarios</i> .

## APÊNDICE D – EXEMPLO DE ARQUIVO SERVERLESS.YML PARA CONFIGURAÇÃO DE APIS

### Trecho de Código D.1 – Exemplo de arquivo *serverless.yml* para configuração da aplicação e nuvem

---

```

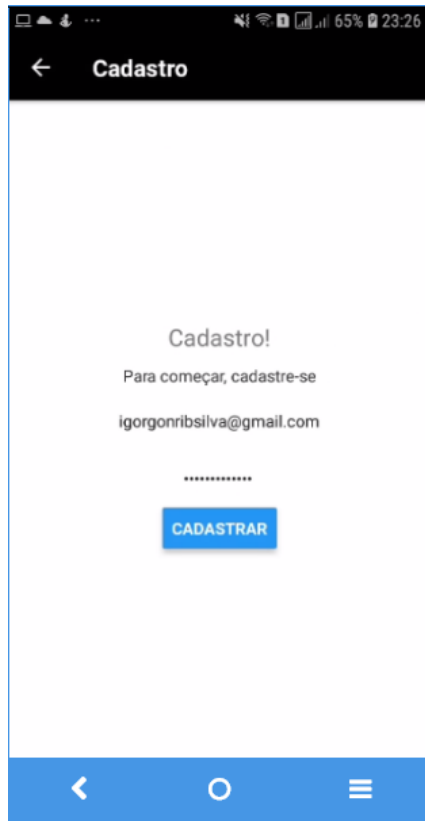
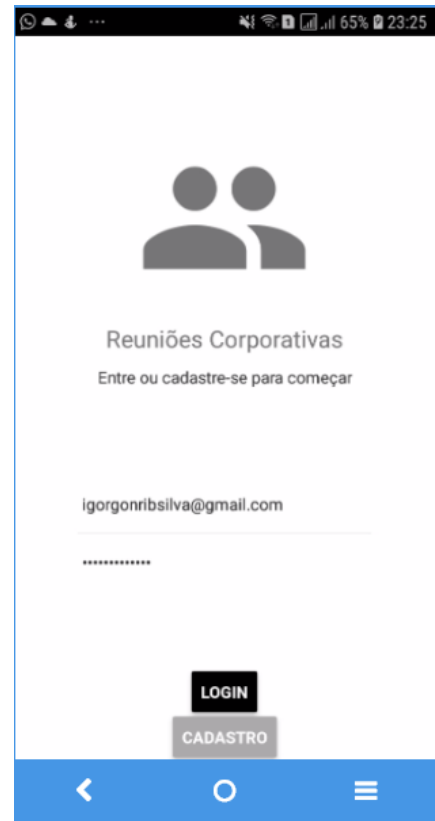
1  #Nome da aplicacao na nuvem
2  service: mobile-serverless-app
3
4  # Plugins utilizados em desenvolvimento
5  plugins:
6    - serverless-webpack
7    - serverless-offline
8
9  # Configuracao de plugins
10 custom:
11   webpack:
12     webpackConfig: ./webpack.config.js
13     includeModules: true
14
15 # Provedor dos servicos de nuvem
16 provider:
17   name: aws
18   runtime: nodejs8.10
19   stage: dev
20   region: us-east-1
21
22 # Configuracoes do IAM
23 iamRoleStatements:
24   # Neste caso a configuracao permite que as funcoes executem as seguintes acoes
25   - Effect: Allow
26     Action:
27       - dynamodb:DescribeTable
28       - dynamodb:Query
29       - dynamodb:Scan
30       - dynamodb:GetItem
31       - dynamodb:PutItem
32       - dynamodb:UpdateItem
33       - dynamodb:DeleteItem
34   # Recursos nos quais as acoes sao permitidas (todos os bancos na regioao us-east-1)
35   Resource: "arn:aws:dynamodb:us-east-1:*:*"
```

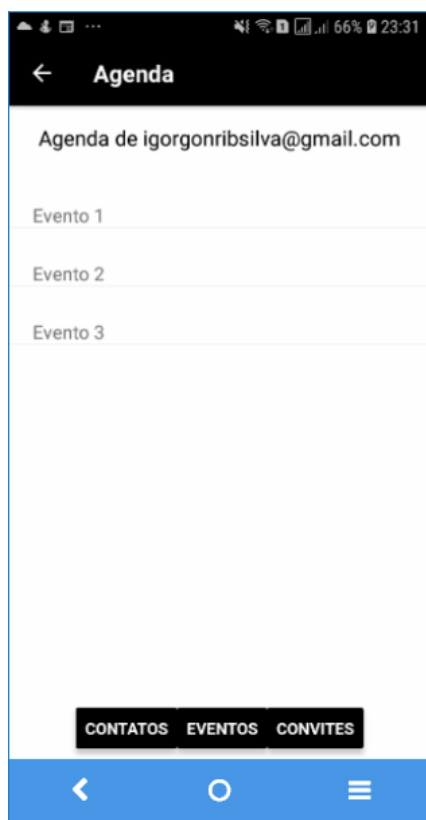
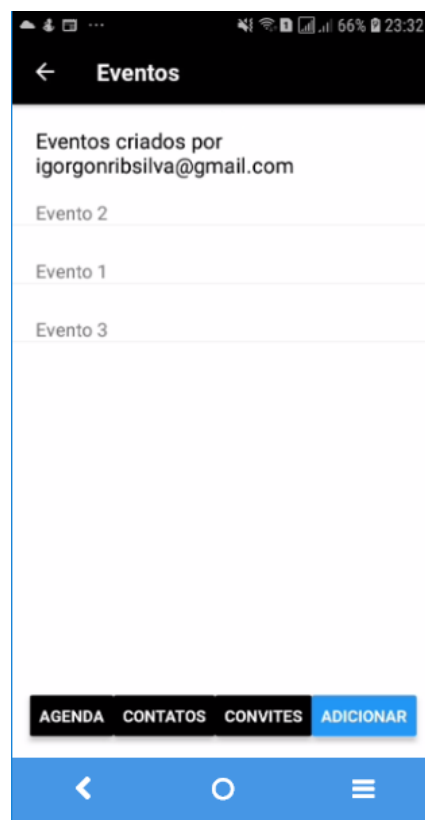
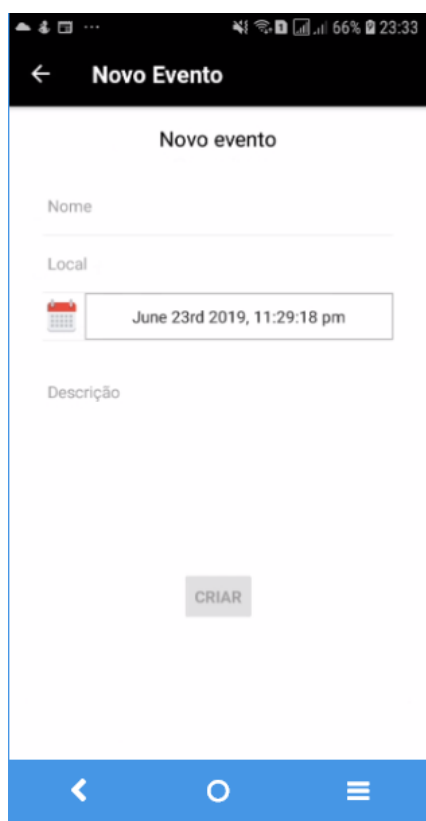
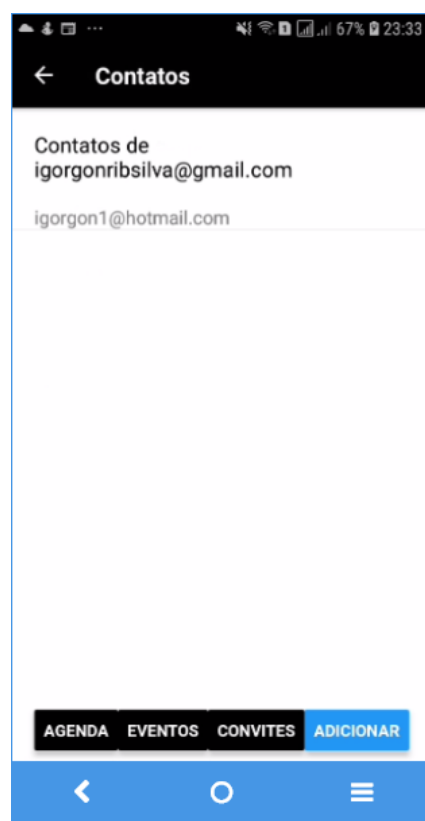


```
36
37 # Funcoes desenvolvidas
38 functions:
39   # Funcao PutUsuario
40   PutUsuario:
41     # Arquivo onde a funcao se encontra
42     handler: handler.PutUsuario
43     # Eventos que disparam a funcao
44     events:
45       - http:
46         # Rota de API para a funcao
47         path: put-usuario
48         # Metodo HTTP suportado
49         method: post
50         # Permite CORS
51         cors: true
52         # Autorizada pelo usuario do IAM configurado
53         authorizer: aws_iam
```

---

## APÊNDICE E – TELAS DO APLICATIVO MOBILE

**Figura 11** – Tela de cadastro**Figura 12** – Tela de login

**Figura 13** – Tela de agenda**Figura 14** – Tela de listagem de eventos**Figura 15** – Tela de criação de eventos**Figura 16** – Tela de listagem de contatos

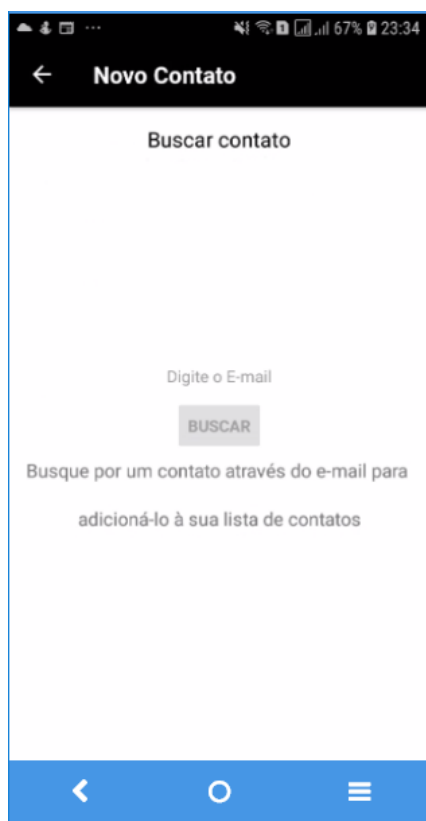


Figura 17 – Tela criação de novo contato

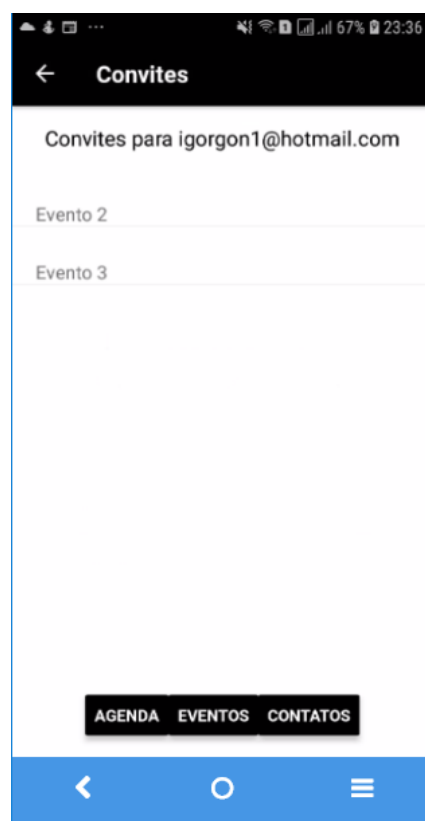


Figura 18 – Tela de listagem de convites



Figura 19 – Tela de visualização de evento

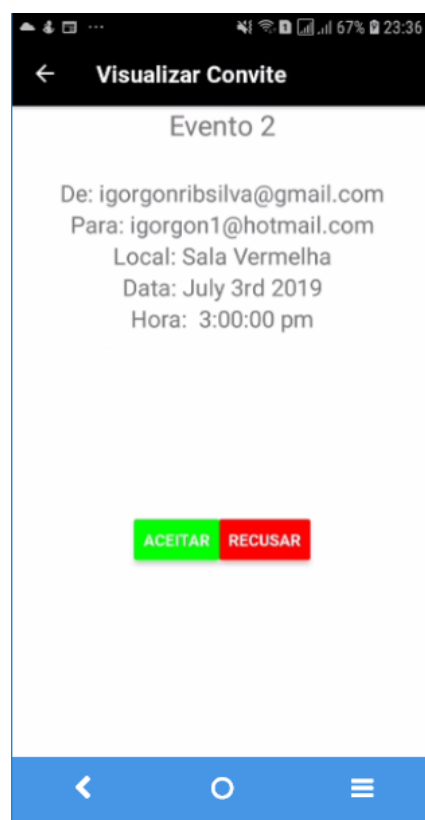


Figura 20 – Tela de visualização de convite