Uma arquitetura de software para sistemas de gerenciamento baseados em IoT

Trabalho de Conclusão de Curso

Engenharia de Computação

Rubens Euclides Carneiro

Orientador: Prof. Dr. Fernando Buarque de Lima Neto

Rubens Euclides Carneiro

Uma arquitetura de software para sistemas de gerenciamento baseados em IoT

Monografia apresentada como requisito parcial para obtenção do diploma de Bacharel em Engenharia de Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Engenharia de Computação

Escola Politécnica de Pernambuco

Universidade de Pernambuco

Orientador: Prof. Dr. Fernando Buarque de Lima Neto

Recife - PE, Brasil

novembro de 2019

Euclides Carneiro, Rubens
Uma arquitetura de software para sistemas de gerenciamento baseados em IoT/ Rubens Euclides Carneiro. – Recife - PE, Brasil, novembro de 2019-
52 p.
Orientador: Prof. Dr. Fernando Buarque de Lima Neto

Trabalho de Conclusão de Curso – Engenharia de Computação
Escola Politécnica de Pernambuco
Universidade de Pernambuco, novembro de 2019.

1. Arquitetura de Software. 2. IoT. 3. Mobile. I. Prof. Dr. Fernando Buarque de Lima Neto. II. Universidade de Pernambuco. III. Escola Politécnica. IV. Título.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

MONOGRAFIA DE FINAL DE CURSO

Avaliação Final (para o presidente da banca)*

No dia 10/12/2019, às 8h30min, reuniu-se para deliberar sobre a defesa da monografia de conclusão de curso do(a) discente **RUBENS EUCLIDES CARNEIRO**, orientado(a) pelo(a) professor(a) **FERNANDO BUARQUE DE LIMA NETO**, sob título Uma arquitetura de software para sistemas de gerenciamento baseados em IoT, a banca composta pelos professores:

LARISSA TENÓRIO FALCÃO ARRUDA (PRESIDENTE)
FERNANDO BUARQUE DE LIMA NETO (ORIENTADOR)

Após a apresentação da monografia e discussão entre os membros da Banca, a mesma foi considerada:

☐ Aprovada ☒ Aprovada com Restrições* ☐ Reprovada

e foi-lhe atribuída nota: 9,0 (nove)

*(Obrigatório o preenchimento do campo abaixo com comentários para o autor)

O(A) discente terá 15 dias para entrega da versão final da monografia a contar da data deste documento.

Larissa Tenório Falcão Arruda

AVALIADOR 1: Prof (a) **LARISSA TENÓRIO FALCÃO ARRUDA**

AVALIADOR 2: Prof (a) **FERNANDO BUARQUE DE LIMA NETO**

AVALIADOR 3: Prof (a)

* Este documento deverá ser encadernado juntamente com a monografia em versão final.

Escola Politécnica de Pernambuco
Graduação em Engenharia de Computação
Projeto de Final de Curso

Autorização de publicação de PFC

Eu, **Rubens Euclides Carneiro** autor(a) do projeto final de curso intitulado: **Uma arquitetura de software para sistemas de gerenciamento baseados em IoT;** autorizo a publicação de seu conteúdo na internet nos portais da Escola Politécnica de Pernambuco e Universidade de Pernambuco.

O conteúdo do projeto de final de curso é de responsabilidade do autor.

Rubens E. Carneiro
Rubens Euclides Carneiro

Orientador(a): **Fernando Buarque de Lima Neto**

Coorientador(a):

Prof, de TCC: **Daniel Augusto Ribeiro Chaves**

Data: 10/12/2019

Resumo

Recentemente houve um crescimento exponencial na disponibilidade de dispositivos móveis com amplo acesso à *internet* e grande poder de processamento, fato que mudou a forma em que as pessoas consomem *softwares*, agora chamados de aplicações ou apps. Junto a esta mudança de paradigma surgiu uma dificuldade para os desenvolvedores, um mesmo sistema precisa ser programado individualmente para diversas plataformas utilizando diversas linguagens de programação. A utilização de *frameworks* multiplataforma visa trazer a agilidade e mantenabilidade do desenvolvimento de *softwares* tradicionais ao mundo das aplicações. A *Internet* das Coisas (IoT) também está em pleno crescimento devido ao barateamento e popularização de novas plataformas de *hardware*. As redes de dispositivos conectados na IoT trazem a necessidade de uma infraestrutura capaz de suportar o grande volume de dados que elas produzem. Sistemas de gerenciamento são um dos principais usos da tecnologia IoT e demandam formas concisas de visualização e meios interação com seus dados. Este projeto definiu uma arquitetura de *softwares* capaz de integrar aplicações móveis multiplataforma a sistemas IoT através da Computação em Nuvem visando a escalabilidade, confiabilidade e mantenabilidade do sistema e a agilidade em seu desenvolvimento. A arquitetura proposta descreveu a infraestrutura necessária para suportar tais sistemas na nuvem e a estrutura do servidor e interface gráfica da aplicação multiplataforma. Um exemplo sem detalhamento está apresentado no final desta monografia.

Palavras-chave: Arquitetura de Software. IoT. Computação em Nuvem. Mobile. Multiplataforma.

Abstract

Recently we had an exponential growth in the availability of mobile devices with wide internet connectivity and high computing power. This changed the way people consume software, which are now called applications or apps. With this paradigm shift, a new problem emerged for developers; a single system needs to be programmed individually for multiple platforms using different programming languages. The use of cross-platform frameworks brings the agility and maintainability of traditional software development to the application world. The Internet of Things (IoT) is also growing due to cost reduction and popularization of new hardware platforms. Device networks connected to the IoT raise the need for an infrastructure capable of handling the large amount of data that they produce. Management systems are one of the main uses of IoT and demand concise visualizations and means of interaction with the data. This project defined a software architecture capable of integrating mobile application with IoT systems through cloud computing while simultaneously considering the scalability, reliability and maintainability of the system and its agile development. This proposed architecture describes the necessary cloud infrastructure to support the system and the structure of both the server and user interface of the cross-platform application. An overview example is provided in the end of this monograph.

Keywords: Software Architecture. IoT. Cloud Computing. Mobile. Multi-platform.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

IoT	*Internet of Things*
IDE	*Integrated Development Environment*
SDK	*Software Development Kit*
HTML	*Hypertext Markup Language*
HTTP	*Hypertext Transfer Protocol*
CSS	*Cascading Style Sheet*
UX	*User Experience*
REST	*Representational State Transfer*
JSON	*Javascript Object Notation*
JWT	*JSON web token*
API	*Application Programming Interface*
MVC	*Model-View-Controller*
ORM	*Object-Relational Mapping*
IaaS	*Infrastructure as a Service*
PaaS	*Platform as a Service*
SaaS	*Software as a Service*
NIST	*National Institute of Standards and Technology*
AWS	*Amazon Web Services*
UID	*Unique Identifier*
CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
RAM	*Random Access Memory*
SSD	*Solid State Disk*

S3	*Simple Storage Service*
RDS	*Relational Database Service*
IAM	*Identity and Access Management*
SNS	*Simple Notification Service*
EC2	*Elastic Compute Cloud*

Sumário

1 Introdução

Este trabalho de conclusão de curso propõe-se ao estudo e desenvolvimento de uma arquitetura de *software* para o desenvolvimento de aplicações móveis multiplataforma para gerenciamento baseado em sistemas de *internet* das coisas.

O crescimento na disponibilidade de sensores e plataformas de *hardware* para *internet* das coisas, e a onipresença de dispositivos móveis na sociedade atual foram alguns dos fatores que desencadearam a escolha deste tema.

Este Capítulo descreve a introdução da monografia, e está organizado em 3 seções. Na Seção 1.1 são descritas tanto a motivação para a execução deste trabalho, quanto a definição do problema.

Posteriormente, nas Seções 1.2 e 1.3 são apresentados os objetivos gerais e específicos, bem como a proposta de solução do projeto. Por fim, a Seção 1.4 detalha a organização da monografia.

1.1 Motivação e Caracterização do Problema

A *Internet* das Coisas, do inglês *Internet of Things* (IoT), é definida como uma infraestrutura global para a sociedade da informação, possibilitando serviços avançados ao interconectar objetos (físicos e virtuais) graças a interoperabilidade de tecnologias da informação e comunicação, atuais e futuras [1].

De acordo com [2], é esperado que em 2022 o investimento em tecnologias IoT alcance $1,2 trilhões com as áreas mais promissoras sendo monitoramento ambiental, medição inteligente e inventário inteligente. Estas áreas podem usufruir de sistemas de gerenciamento onde seja possível visualizar os dados monitorados e realizar decisões, que por sua vez pode ser feita através de comandos enviados aos objetos da IoT. Atualmente uma das quatro maiores dificuldades da adoção da tecnologia por empresas é a falta de infraestrutura tecnológica para integrar os sistemas existentes às redes IoT.

Uma das dimensões que necessita apoio computacional é justamente um mecanismo para engajar o usuário final nos sistemas que utilizam Inteligência Artificial. Portanto, é importante que a interface gráfica de aplicações de gerenciamento com integração a sistemas IoT sejam acessíveis e intuitivas. Atualmente estes requisitos são pouco atendidos em dispositivos móveis como celulares e *tablets*.

Desenvolver uma solução voltada a essa tecnologia se torna portanto essencial visto que o uso diário médio de dispositivos móveis está aumentando a cada ano, principalmente

entre usuários na faixa de idade entre 16 e 24 anos. Neste ano, por exemplo, deve haver um volume de $120 bilhões em compras de aplicações móveis (*mobile apps*) [3].

Devido a diversidade de dispositivos disponíveis no mercado, um problema recorrente é a necessidade de codificar separadamente uma mesma aplicação para cada plataforma [4]. Esta situação dificulta a evolução de empresas já que demanda mais tempo ou capital humano para o desenvolvimento e manutenção da aplicação. Nesta monografia de TCC vamos contribuir justamente com a proposição de uma arquitetura para dispositivos móveis que além de intuitiva, atenda aspectos de IoT, e não necessite de re-escrita para diferente plataformas através da utilização de *frameworks* multiplataforma.

1.2 Objetivo Geral

Este projeto tem como objetivo propor uma arquitetura de *software* para o desenvolvimento de aplicações móveis multiplataforma para gerenciamento baseado em sistemas IoT. A arquitetura definida deve proporcionar o desenvolvimento ágil e a mantenabilidade do sistema, notadamente sua evolução funcional. A proposta define a infraestrutura necessária, a organização do servidor de dados (*back-end*) e a interface de usuário (*front-end*).

1.3 Objetivos Específicos

A implementação do projeto também envolveu outros fins, os quais foram atingidos:

- A especificação da infraestrutura que permite a integração do sistema IoT ao *software*;
- A especificação da arquitetura do *back-end* do *software*;
- A especificação da arquitetura do *front-end* do *software* utilizando uma abordagem multiplataforma;

1.4 Estrutura do Documento

Este trabalho está dividido em 5 Capítulos, incluindo este que conta com uma introdução a respeito do tema e objetivos do projeto. Em seguida o Capítulo 2 traz um estudo a respeito de arquitetura de *software*, desenvolvimento de aplicações móveis e demais conceitos necessários para a compreensão da arquitetura desenvolvida neste trabalho.

Já no Capítulo 3, é descrito as etapas do processo que resultou na arquitetura proposta. O Capítulo 4 apresenta os resultados obtidos.

Finalmente, o Capítulo 5 encerra o trabalho com uma discussão a respeito do que foi desenvolvido, possíveis melhoramentos e novas ideias para o projeto.

2 Fundamentação Teórica

Este Capítulo busca apresentar alguns conceitos fundamentais ao desenvolvimento da arquitetura proposta. Na Seção 2.1 serão abordadas as definições de *internet* das coisas. Em seguida, a Seção 2.2 introduz a computação em nuvem. A Seção 2.3 apresenta os dispositivos móveis que serão atingidos pelo projeto. Por fim, na Seção 2.4 serão discutidos as arquiteturas de *software* em que o projeto se baseia.

2.1 *Internet* das Coisas

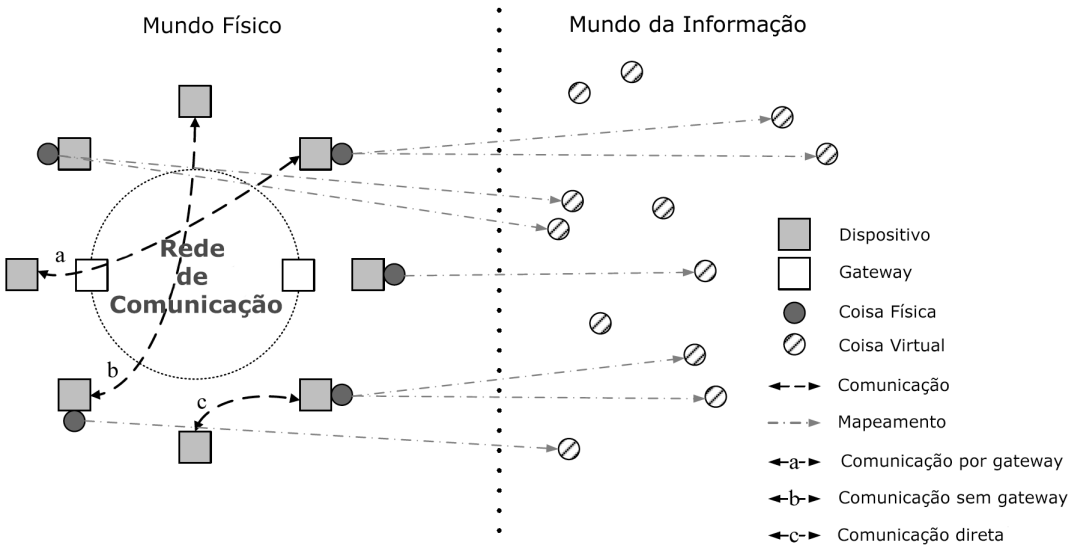
Extendendo a definição que utilizamos na Seção 1.1, a *Internet* das Coisas, do inglês *Internet* of things (IoT), é um sistema composto por computadores, maquinas mecânicas ou digitais, objetos, animais ou pessoas que possuem um identificador único, do inglês *unique identifier* (UID), e a habilidade de transmitir dados através de uma rede sem a necessidade de interação humano-humano ou humano-maquina [5].

As Coisas são objetos do mundo físico (Coisas físicas) ou do mundo da informação (Coisas virtuais) que são capazes de serem identificadas e integradas a redes de comunicação. Coisas físicas são capazes de serem sentidas, atuadas e conectadas, alguns exemplos são o ambiente, robôs industriais, bens e equipamentos elétricos. Coisas virtuais são capazes de serem armazenadas, processadas e acessadas, exemplos incluem conteúdos *multimedia* e aplicações de *software* [1].

Uma Coisa física pode ser representada no mundo da informação por uma ou mais Coisas virtuais através de mapeamentos. Dispositivos podem se comunicar de várias formas, através de uma rede por meio de um *gateway* (caso a), através de uma rede sem um *gateway* (caso b) e diretamente (caso c).

Redes de comunicação transmitem de forma confiável e eficiente os dados gerados ou captados pelos dispositivos a aplicações ou outros dispositivos, como também instruções enviadas por aplicações a dispositivos. A figura 1 é uma visualização destes conceitos.

Figura 1 – Visão técnica da IoT.



Fonte: adaptado de (ITU-T, 2012).

2.1.1 Características

As redes de dispositivos IoT apresentam aspectos distintos em relação a outras redes de comunicação, como por exemplo redes celulares, permitindo sua utilização em contextos de escala pessoal até industrial. Abaixo, temos uma visão sobre as características da *Internet* das Coisas que nos permite entender sua capacidade e desafios [1].

- **Interconectividade**: Na IoT, tudo pode se conectar a infraestrutura global de informação e comunicação.
- **Serviços**: A IoT é capaz de prover serviços relacionado às Coisas de acordo com suas restrições, como privacidade e consistência entre a semântica das Coisas físicas e Coisas virtuais. Para prover novos serviços, as tecnologias no mundo físico e no mundo da informação precisam se adaptar.
- **Heterogeneidade**: Dispositivos na IoT são heterogêneos por serem baseados em diferentes plataformas de *hardware*. Podendo interagir com outros dispositivos ou plataformas de serviços através de diferentes redes.
- **Mudanças dinâmicas**: O estado dos dispositivos mudam dinamicamente, podendo por exemplo estar adormecidos, ativos, conectados ou desconectados, além de mudanças em contextos como localização e velocidade.

- **Enorme escala**: O número de dispositivos que precisam ser gerenciados e que se comunicam estarão em ao menos uma ordem de grandeza maior que dispositivos atualmente conectados à *internet*. Ainda mais crítico é o gerenciamento dos dados gerados e sua interpretação para aplicações.

2.1.2 Requisitos de Alto Nível

Para o funcionamento eficaz de uma rede de dispositivos IoT é importante que haja uma estruturação adequada da infraestrutura física e de *software* para que os seguintes requisitos de alto nível sejam atendidos, de acordo com o contexto desejado [1].

- **Conectividade baseada em identificação**: A conexão entre uma Coisa e a IoT deve ser baseada no identificador da Coisa. Possivelmente, identificadores heterogêneos podem ser processados de uma forma unificada.
- **Interoperabilidade**: A interoperabilidade precisa ser garantida entre diferentes dispositivos e sistemas para a produção e consumo de informações e serviços.
- **Provisionamento autônomo de rede**: Técnicas ou mecanismos autônomos de gerenciamento, configuração, recuperação, otimização e proteção devem ser suportados pela IoT para que se adapte a diferentes domínios de aplicação, diferentes ambientes de comunicação e ao grande número e tipos de dispositivos.
- **Provisionamento autônomo de serviços**: Os serviços precisam ser providos a partir da captura, comunicação e processamento automático dos dados das Coisas, baseando-se em regras configuradas pelos operadores ou usuários finais.
- **Localização**: A IoT deve suportar comunicações e serviços dependentes da localização de Coisas e seus usuários. Estas operações podem ser restritas por leis e regulações, além de que precisam aderir a requisitos de segurança.
- **Segurança**: Na IoT todas as Coisas estão conectadas, desta forma surgem ameaças de segurança sobre a confidencialidade, autenticidade e integridade dos dados e serviços. Serviços autônomos podem depender de técnicas automáticas como *data fusion* e mineração de dados.
- **Privacidade**: Proteção a privacidade precisa ser suportado pela IoT. Dados adquiridos pelas Coisas podem conter informações privadas sobre seus donos ou usuários. A proteção precisa estar presente na transmissão, agregação, armazenamento, mineração e processamento dos dados.

- **Serviços baseados no corpo humano**: Serviços e segurança de alta qualidade precisam ser suportados para serviços baseados no corpo humano. Tais serviços incluem a captura, comunicação e processamento de dados relacionados a características do corpo humano e seu comportamento. Países possuem leis e regulações diferentes para estes serviços.
- **Gerenciamento**: A capacidade de gerenciamento precisa ser suportada na IoT para garantir o funcionamento das operações de rede. Aplicações IoT normalmente trabalham automaticamente, mas o processo deve ser gerenciável pelas pessoas envolvidas.

2.2 Computação em Nuvem

Computação em nuvem, do inglês *cloud computing*, é um modelo que permite acesso de rede ubíquo, conveniente e sob demanda a um conjunto compartilhado de recursos computacionais, como redes, servidores, armazenamento, aplicações e serviços, que podem ser rapidamente alocados ou liberados com o mínimo de esforço de gerenciamento ou interferência do provedor do serviço. Este modelo de computação em nuvem é composto por cinco características e três modelos de serviço [6].

2.2.1 Características

As cinco características de computação em nuvem definidas pelo Instituto Nacional de Padrões e Tecnologia dos EUA (NIST) [6] são:

- **Serviços sob demanda**: Um consumidor pode provisionar unilateralmente recursos computacionais, como tempo de servidor e armazenamento, sem requerer interação humana com cada provedor de serviços.
- **Amplo acesso à rede**: Recursos estão disponíveis pela rede e podem ser acessados através de mecanismos que promovem seu uso por plataformas de cliente, como dispositivos móveis e computadores.
- **Conjunto de recursos**: Os recursos computacionais do provedor são compartilhados para atender múltiplos consumidores utilizando um modelo *multi-tenant*, com diferentes recursos físicos e virtuais dinamicamente alocados e realocados de acordo com a demanda do consumidor. Exemplos de recursos incluem armazenamento, processamento, memória e largura de banda para a rede.
- **Elasticidade**: Recursos podem ser provisionados e liberados elasticamente, em alguns casos automaticamente, para escalar externamente e internamente de acordo com a demanda.

- **Serviço de medição**: Sistemas em nuvem controlam e otimizam automaticamente o uso de recursos ao utilizarem ferramentas de medição. O uso de recursos pode ser monitorado, controlado e reportado oferencendo transparência ao provedor e ao consumidor do serviço utilizado.

2.2.2 Modelos de Serviço

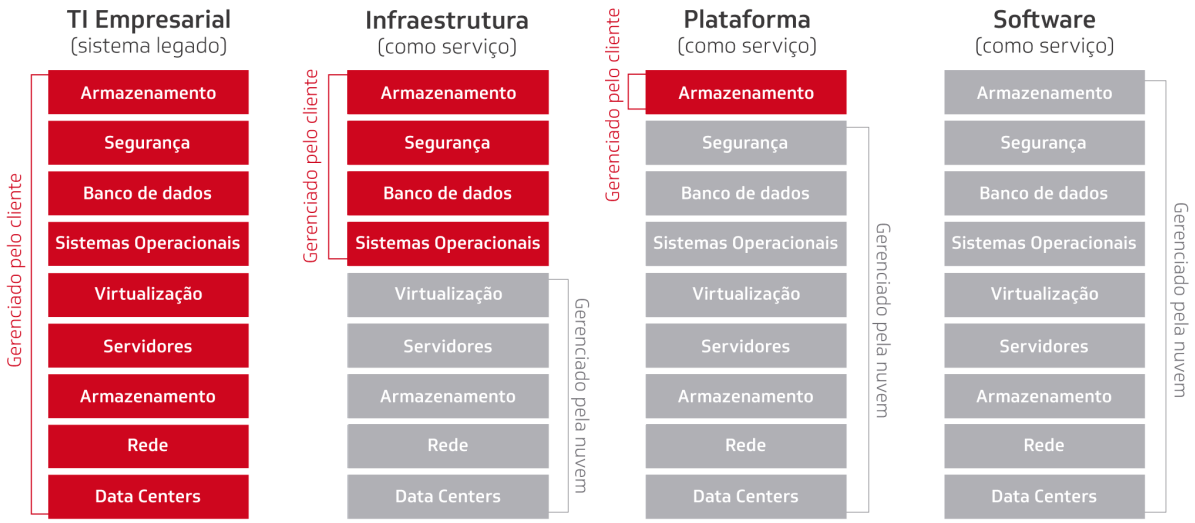
Provedores de computação em nuvem comumente fornecem serviços divididos em diferentes camadas para atender necessidades específicas dos consumidores, já que diferentes níveis de controle sobre a infraestrutura podem ser necessários. A figura 2 trás uma comparação entre os diferentes modelos de serviço que serão discutidos. O NIST define estes modelos de serviço [6] como:

- ***Software* como serviço**, do inglês *Software as a Service* (SaaS), é um modelo em que o consumidor deve utilizar aplicações fornecidas pelo provedor que são executadas numa infraestrutura de nuvem. As aplicações são acessíveis através de interfaces de usuário, como navegadores de *internet*, ou interfaces de programação, do inglês *Application Programming Interface* (API). O consumidor não gerencia ou controla a infraestrutura contratada, sendo limitado apenas a configurações específicas da aplicação.

- **Plataforma como serviço**, do inglês *Platform as a Service* (PaaS), é um modelo que permite que o consumidor implante aplicações próprias ou adquiridas que foram criadas utilizando linguagens de programação, bibliotecas, serviços e ferramentas compatíveis com a infraestrutura de nuvem do provedor. O consumidor não gerencia ou controla a infraestrutura contratada, mas tem controle sobre as aplicações implantadas e possivelmente a configurações do ambiente que hospeda tais aplicações.

- **Infraestutura como serviço**, do inglês *Infrastructure as a Service* (IaaS), é um modelo onde o consumidor é capaz provisionar processamento, armazenamento, rede e outros recursos computacionais fundamentais para implantar e executar *softwares* arbitrários, incluindo sistemas operacionais e aplicações. O consumidor não gerencia ou controla a infraestrutura física, mas tem controle sobre sistemas operacionais, armazenamento, aplicações e, possivelmente, componentes de rede como *firewalls*.

Figura 2 – Modelos de serviços da nuvem.



Fonte: extraído de (Solo Network, 2019).

2.2.3 Plataformas

Os serviços necessários para a estruturação da arquitetura proposta estão disponíveis na maioria das plataformas de nuvem, como a *Amazon Web Services*, *Microsoft Azure*, *Google Cloud* e *Alibaba Cloud*. Devido aos altos índices de disponibilidade, escalabilidade e performance aliados a um baixo custo inicial foi escolhida a nuvem da *Amazon Web Services* como infraestrutura para a prova de conceito desenvolvida neste projeto. Sua base de clientes envolve milhões de usuários ativos todos os meses e incluem nomes como a NASA, Netflix, Uber, Formula 1, BMW, Penn State, McDonalds e muitos outros.

A ***Amazon Web Services*** (AWS) é uma subsidiaria da *Amazon.com, Inc* que fornece plataformas de computação em nuvem e APIs sob demanda para pessoas, empresas e governos. Os *web services* oferecidos disponibilizam um conjunto abstrato de infraestrutura física, bases para computação distribuída e ferramentas.

A tecnologia da AWS é implementada em fazendas de servidores espalhadas pelo mundo. O custo de utilização é baseado numa combinação de uso, componentes de *hardware*, *software* e rede contratados, disponibilidade necessária, redundância e segurança. Assinantes podem pagar por um único computador virtual da AWS, um computador físico dedicado ou um conjunto, *cluster*, de ambos.

Atualmente a AWS fornece 165 serviços que abrangem computação, armazenamento, redes, bancos de dados, *analytics*, aprendizado de máquinas, aplicações, implantação e gerencia de sistemas, ferramentas de desenvolvimento entre outros. Muitos destes serviços estão disponíveis a partir de APIs acessíveis através de HTTP utilizando a arquitetura REST que será discutida em seguida.

2.3 Dispositivos Móveis

Dispositivos móveis, *mobile* ou *handheld* em inglês, são dispositivos de computação pequenos o suficiente para serem segurados e operados usando as mãos. Tipicamente estes dispositivos possuem acesso a *internet* e pode se conectar a outros aparelhos utilizando tecnologias como Wi-Fi, Bluetooth e redes celulares. Estes dispositivos são produzidos por diversas empresas e cada um possui estruturas de hardware e software diferentes [8].

Os dispositivos mais comuns são *smartphones*, *tablets* e *smartwatches*, mas levando a definição ao pé da letra também temos nesta lista aparelhos como *laptops*, calculadoras, *videogames* portáteis e assistentes pessoais (PDA). A arquitetura proposta neste projeto será voltada a *smartphones* já que este é o dispositivo mais utilizado atualmente [3].

Cada *smartphone* possui um sistema operacional que é a interface entre as aplicações e as propriedades do hardware do dispositivo como a tela sensível ao toque, camera, conectividade, GPS e armazenamento. Atualmente o Android e iOS possuem juntos uma fatia de 98% do mercado de sistemas operacionais para dispositivos móveis, portanto elas serão as plataformas consideradas durante o projeto [9].

2.3.1 Plataforma iOS

iOS é o sistema operacional para dispositivos móveis desenvolvido pela *Apple Inc.* exclusivamente para o iPhone, iPad e iPod.

O desenvolvimento de aplicações para iOS é realizado utilizando o pacote de desenvolvimento de *software*, do inglês *software development kit* (SDK), XCode também fornecido pela *Apple Inc.*. Através deste SDK é possível criar, testar e distribuir aplicações para iOS utilizando as linguagens *Swift* ou *Objective-C*.

2.3.2 Plataforma Android

Android é um sistema operacional para dispositivos móveis baseado em uma versão do núcleo do Linux e outros *softwares* de código aberto. Seu desenvolvimento é realizado pelo consorcio *Open Handset Alliance*, que tem o *Google LLC* como principal contribuidor.

Aplicações para Android são produzidas utilizando o Android SDK, pacote que inclui emuladores de dispositivos, bibliotecas de *softwares*, depuradores, documentações

e guias. O ambiente de desenvolvimento, do inglês *Integrated Development Environment* (IDE), oficial para o desenvolvimento de aplicações Android é o AndroidStudio, que é capaz de criar e testar aplicações escritas em Java, Kotlin ou C++.

2.3.3 Multiplataforma

Devido a diversidade de dispositivos disponíveis no mercado, um problema recorrente é a necessidade de codificar separadamente uma mesma aplicação para cada plataforma. Para mitigar este problema foram criados *frameworks* multiplataforma, conjunto de classes que permitem a construção de aplicações multiplataforma com pouco esforço, especificando apenas as particularidades de cada aplicação. Desta forma é alcançada uma maior agilidade no desenvolvimento da aplicação já que apenas um código precisa ser escrito para executar em ambas plataformas, consequentemente beneficiando a mantenabilidade do sistema.

O ***React Native*** é um *framework* de código aberto desenvolvido pelo *Facebook* para o desenvolvimento de aplicações móveis multiplataforma utilizando o *React*, uma biblioteca *Javascript* para criação de interfaces de usuários. Esta tecnologia permite o completo desenvolvimento de aplicações utilizando apenas *Javascript*, ao invés de *Objective-C* para a plataforma iOS e *Java* para plataforma Android.

As interfaces de usuário criadas em *React Native* são compostas por componentes. Estes componentes são criados utilizando uma extensão da sintaxe do Javascript chamada *Javascript XML* (JSX), uma combinação de código *Javascript* com uma linguagem de marcação semelhante a HTML. Desta forma o desenvolvimento de aplicações móveis se torna bastante semelhante à criação de *web pages*. A figura 3 mostra uma aplicação desenvolvida com React Native sendo executada nas plataformas iOS e Android.

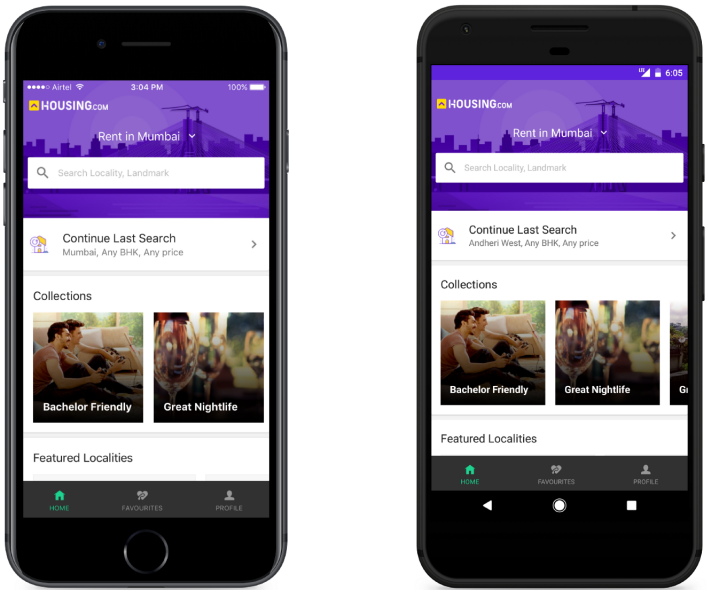
2.4 Arquitetura de *Software*

A arquitetura de *software* fornece uma visão holística do sistema a ser construído. Ela representa a estrutura e organização dos componentes de *software*, suas propriedades e as conexões entre eles. Os componentes de *software* incluem módulos de programas e as várias representações de dados manipuladas pelo programa [11].

2.4.1 Padrões Arquiteturais

Os padrões arquiteturais são guias para resolução de problemas recorrentes a fim de permitir uma estruturação eficaz de sistemas. Os padrões que serão utilizados neste projeto serão discutidos.

Figura 3 – Aplicação executando nas plataformas iOS e Android, respectivamente.



Fonte: extraído de (Guerra, 2015).

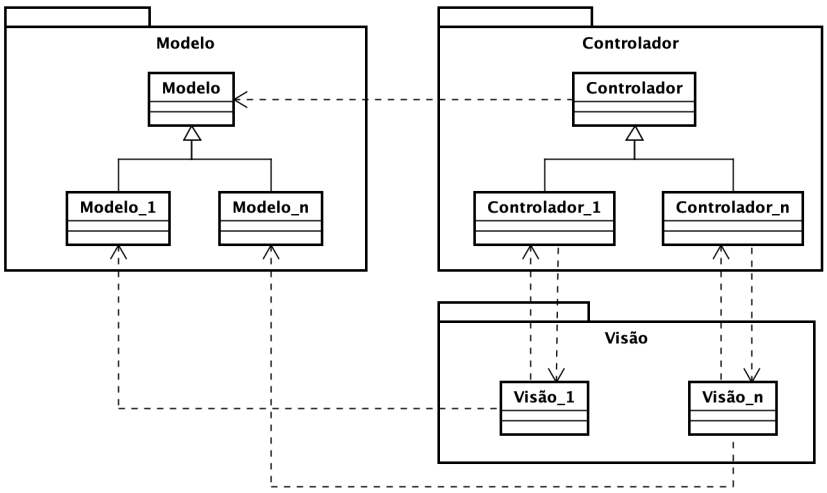
2.4.2 *Model-View-Controller*

O padrão *Model-View-Controller* divide uma aplicação interativa em três camadas. O Modelo, *Model*, possui as funcionalidades principais do sistema e os dados. A Visão, *View*, exibe informações ao usuário. O Controlador, *Controller*, trata das entradas do usuário. As visões e controladores fazem parte da interface de usuário. Um mecanismo de propagação de mudança garante a consistência entre a interface e o modelo [12].

O diagrama da figura 4 mostra as relações entre as camadas do padrão MVC e a figura 5 demonstra o fluxo de dados de uma aplicação móvel que utiliza este padrão. Um usuário inicia sua interação através de uma Visão, como uma interface gráfica, que trata este evento e aciona o Controlador correspondente. Por sua vez, o Controlador realiza uma consulta ou requisita uma modificação ao Modelo, que interage com o banco de dados e notifica o Controlador e a Visão sobre mudanças. O controlador pode então manusear a resposta do Modelo e enviar o resultado à Visão para ser exibida ao usuário.

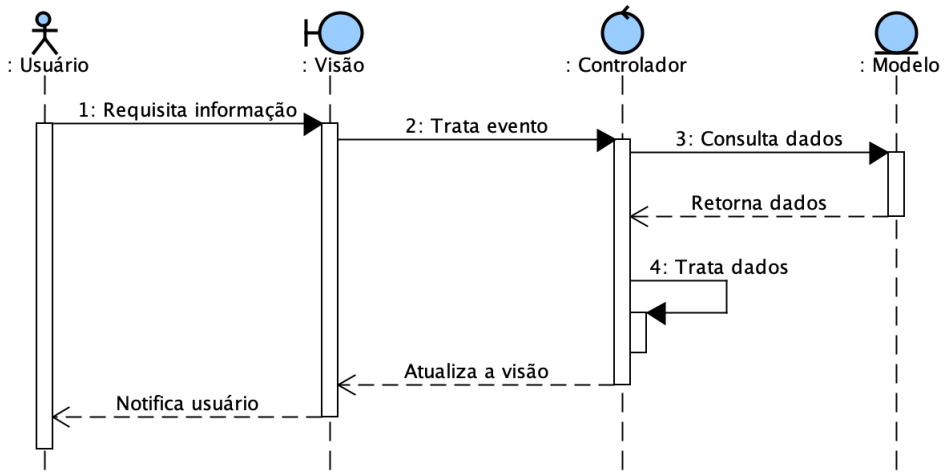
A utilização do MVC trás vantagens interessantes a sistemas interativos, a separação entre o Modelo e a interface de usuário permite que várias Visões diferentes possam ser implementadas e usem um mesmo Modelo. Esta separação permite a modularização da interface de usuário, permitindo até a substituição de componentes em *run-time*. Outra vantagem é o mecanismo de propagação de mudanças que garante que todos os elementos dependentes sejam notificados sobre mudanças nos dados da aplicação, mantendo assim a coerência do sistema [12].

Figura 4 – Diagrama UML do padrão MVC.



Fonte: o autor.

Figura 5 – Diagrama de sequência de um sistema interativo que utiliza o padrão MVC.



Fonte: o autor.

2.4.3 *Representational State Transfer*

Representational State Transfer (REST), é uma arquitetura de software que define um conjunto de restrições para a criação de *web services*, soluções utilizadas para permitir a integração de sistemas através de uma rede.

A *internet* opera como um sistema de informação que impõe várias restrições: Agentes identificam objetos no sistema, chamados de recursos, através de identificadores uniformes de recursos, do inglês *Uniform Resource Identifier* (URI).

Agentes representam, descrevem e comunicam o estado de um recurso através de representações em vários formatos de dados, como por exemplo XML, HTML, CSS, JPEG, PNG. Agentes trocam representações através de protocolos que utilizam URIs para identificar e acessar diretamente ou indiretamente os agentes e recursos [13].

A rede REST é um sub conjunto da *internet* em que agentes utilizam uma interface semântica uniforme, que essencialmente cria, busca, atualiza e remove recursos, ao invés de interfaces arbitrárias ou especificas a uma aplicação. Os recursos são manipulados através da troca de representações. Além disso as interações REST são *stateless*, ou seja, o significado da mensagem não depende do estado da conversação [13].

Formalmente as restrições do REST, definidas por [14], são:

- **Arquitetura cliente-servidor**: Separar as responsabilidades da interface de usuário das responsabilidades do armazenamento de dados aumenta a portabilidade da interface de usuário para multiplas plataformas.
- ***Statelessness***: A comunicação cliente-servidor fica restringida a não armazenar contexto do cliente no servidor entre requisições. Cada requisição do cliente possui toda informação necessária para completar a operação requisitada.
- ***Cacheability***: O armazenamento temporário, do inglês *Caching*, de respostas do servidor pode ser realizado pelo cliente a fim de diminuir a interação cliente-servidor, desta forma melhorando a eficiência da rede.
- **Sistema em camadas**: Estrutura o sistema em camadas para permitir a utilização de balanceadores de carga e a separação de componentes por questões semânticas ou segurança.
- **Interface uniforme**: Os quatro elementos que definem a interface são recursos identificados a partir de URIs; a manipulação de recursos feita através de representações; mensagens que possuem toda informação necessária para seu processamento; *links* providos pelo servidor para que o cliente possa descobrir todas as ações e recursos disponíveis.

3 Arquitetura Proposta

Este Capítulo busca explicar os passos que envolvem o desenvolvimento de cada módulo da arquitetura de software proposta. O projeto utiliza uma abordagem *bottom-up*, começando a partir dos sistemas físicos até a aplicação em *software*, para tornar mais evidente os requisitos de cada módulo do sistema. Portanto, segue as seguintes etapas:

I Definição da infraestrutura necessária para a integração entre o sistema IoT e o *software*: A Seção 3.1 explora como a interface IoT-*Software* é organizada.

II Definição da estrutura do *back-end* do *software*: Na Seção 3.2 são especificadas as tecnologias relevantes para o funcionamento do servidor e sua integração com os dados e a interface de usuário.

III Definição da estrutura do *front-end* do *software* utilizando uma abordagem multiplataforma: Por fim a Seção 3.3 especifica as tecnologias necessárias para o desenvolver aplicações *mobile* multiplataformas.

3.1 Infraestrutura

Para garantir que o sistema atenda requisitos de disponibilidade e confiabilidade é utilizada uma plataforma de computação em nuvem utilizando o modelo de serviço IaaS, discutido na Seção 2.2.2. Desta forma os desenvolvedores do sistema não precisam construir e gerenciar a estrutura física dos servidores, podendo focar apenas em sua configuração, além disso o custo de implantar e manter um infraestrutura *on-premise*, em muitos casos, é bem maior do que contratar uma infraestrutura em nuvem.

3.1.1 Acesso à Nuvem

Usuários conectam-se através de sistema de nomes de domínio em nuvem, do inglês *Domain Name System* (DNS), de alta disponibilidade e escalabilidade. Este serviço é projetado para entregar aos desenvolvedores uma forma confiável e de bom custo beneficio de rotear usuários finais para aplicações na internet [15]. Através deste serviço é possível criar um nome de domínio próprio na rede interna da nuvem, para que os usuários possam acessar seus sistemas através de endereços como "www.empresa.com.br"ao invés de utilizar um endereço de nuvem como "ec2-33-212-216-103.compute-1.amazonaws.com".

Este serviço também é utilizado para escalar aplicações do tipo *white-label*, no qual um mesmo produto pode ser vendido para diferentes clientes com apenas mudanças estéticas. Neste contexto é possível criar diferentes nomes de domínio para um mesmo serviço, permitindo que ele seja acessível através dos endereços "www.empresaA.com.br"e "www.empresaB.com.br", por exemplo.

Para controlar o acesso ao sistema, tanto de clientes ou funcionários, deve ser utilizado um serviço de controle de acesso. Ele permite o gerenciamento do acesso aos recursos e serviços da nuvem de forma segura através da criação de usuários e grupos, efetivamente permitindo ou negando o acesso a partir de um conjunto permissões [16]. Tal serviço é importante para que requisitos de segurança sejam atendidos, já que é possível, por exemplo, limitar a visualização de dados nos sistemas de armazenamento utilizados.

É necessário um serviço de integração de dispositivos IoT com a plataforma de nuvem que permita conexões através dos protocolos HTTP, *WebSockets*, MQTT e protocolos específicos a dispositivos. Um sistema de identificação semelhante ao de controle de acesso de usuários é utilizado para proteger a conexão entre os dispositivos e a nuvem além de autenticar o envio de comandos, certificando-se que apenas usuários autorizados possam interagir com os aparelhos e visualizar seus dados [17].

O estado mais recente de um dispositivo conectado pode ser automaticamente armazenado por este serviço para que possa ser lido e modificado a qualquer momento, fazendo com que o dispositivo sempre pareça conectado à aplicação. Desta forma é possível ler a informação mais recente e enviar comandos a dispositivos desconectados, no qual serão entregues quando a conexão for reestabelecida seguindo regras pre-estabelecidas pelos desenvolvedores.

3.1.2 Armazenamento

Plataformas de nuvem fornecem serviços de armazenamento de dados em diversos formatos. A natureza de sistemas IoT trás a necessidade da armazenar os dados em duas formas, em objetos e em banco de dados. Desta forma podemos dividir a carga de trabalho necessária para a geração de dados estruturados.

Armazenamento de objetos é uma arquitetura de armazenamento de dados que trata os dados como objetos, em contraste as arquiteturas de sistemas de arquivos ou armazenamento em blocos. Cada objeto possui um UID global, metadados e os dados em si. Essa estrutura permite a utilização de interfaces que podem ser programadas pela aplicação, funções de gerenciamento como replicação e distribuição de dados em alta granularidade e espaços de nomes, *namespaces*, que pode existir em mais de um *hardware* físico [18].

Os dispositivos IoT podem realizar simples pre-processamentos e enviar os dados brutos para o sistema de armazenamento de objetos, a partir disso um sistema com mais poder computacional pode processar estes dados e estrutura-los para o banco de dados ou ferramentas de mineração de dados. Devido a infraestrutura das principais nuvens, a transferência de dados entres os sub-sistemas é bastante eficiente, podendo chegar a taxas de transmissão de até 100 Gbps.

Deve ser utilizado um sistema de armazenamento de objetos que possa armazenar informações em qualquer formato. Os consumidores, sejam usuários ou equipamentos IoT, conectam-se ao sistema e transferem dados utilizando uma conexão direta através de uma API. Os casos de uso comuns de tais serviços cobrem armazenamento para aplicações, *backups* e recuperação, arquivamento e *data lakes* para análise de dados. [19].

Serviços de armazenamento de objetos podem ser personalizado de acordo com as necessidades do usuário, para tal são oferecidas classes de armazenamento para acesso frequente; armazenamento longo com acesso infrequente; e para arquivamento de longo prazo e preservação digital. Objetos podem ser migrados dinamicamente entre estas classes de acordo com seu uso, caso o usuário deseje.

Um serviço de banco de dados distribuído é essencial para o funcionamento do sistema. Estes serviços são oferecidos através de *web services* desenvolvidos para simplificar a configuração, operação e escalabilidade de bancos de dados relacionais para aplicações. Processos administrativos como provisionamento de *hardware*, atualizações, *backups* e recuperações são gerenciados pela plataforma de nuvem. Semelhante ao sistema de armazenamento de objetos, o serviço pode ser configurado através de chamadas de API. Os bancos de dados suportados, com otimizações de memória, performance e E/S, geralmente incluem o PostgreSQL, MySQL, Oracle Database, MongoDB e CassandraDB. [20].

3.1.3 Computação

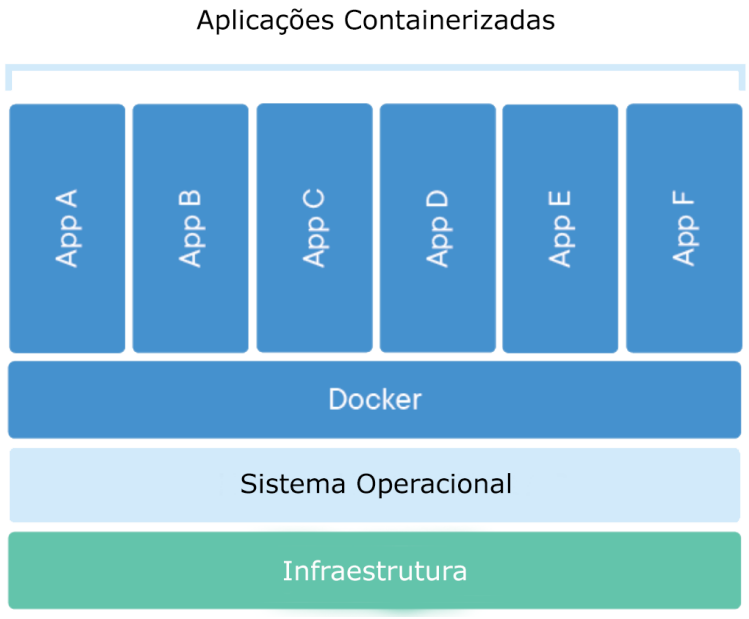
Serviços de computação em nuvem permitem que usuários disponham de um *cluster* de computadores disponíveis a todo momento através da *internet*. As máquinas virtuais da nuvem possuem atributos que podem ser personalizadas de acordo com a necessidade do usuário como quantidade de CPUs e GPUs, memória RAM, discos rígidos ou SSD, sistemas operacionais, rede e aplicações pre-instaladas como servidores *web* e banco de dados [21].

Para aprimorar a mantenabilidade e escalabilidade do sistema, dentro da plataforma pode ser utilizado o Docker, um sistema que gerencia *softwares* em pacotes chamados de *containers*. Um *container* é uma unidade de *software* que une código e dependências para que a aplicação funcione de maneira previsível e confiável em diferentes ambientes.

Múltiplos *containers* podem ser executados no mesmo computador e compartilhar o *kernel* do sistema operacional, onde cada um é executado em processos isolados e com canais bem definidos de comunicação. Devido a sua arquitetura, *containers* ocupam menos espaço que máquinas virtuais e podem suportar mais aplicações [22].

Uma imagem de *container* é um pacote de *software* leve, independente e executável que inclui tudo que é necessário para executar uma aplicação: código, *runtime*, ferramentas do sistema, bibliotecas e configurações. Imagens são o método principal de armazenamento e distribuição de aplicações "containerizadas". Desta forma o servidor pode ser dividido em diversas aplicações, como por exemplo APIs, servidores *web* e servidores de *cache*, que podem se comunicar através de uma rede interna. A figura 6 mostra a organização de um servidor que utiliza o Docker executando diversas aplicações.

Figura 6 – Estrutura do Docker



Fonte: adaptado de (Docker, 2019).

3.1.4 Notificação

Um dos requisitos principais em uma aplicação de gerenciamento é a capacidade de notificar o usuário sobre o acontecimento de algum evento. Esta notificação pode ser realizada de diversas formas, em dispositivos móveis é comum que o envio seja através de *push notifications*, *emails* ou mensagens SMS.

Serviços de mensagem pub/sub com alta disponibilidade e segurança permitem desacoplar microsserviços, sistemas distribuídos e aplicações *serverless*. A utilização de tópicos, conjunto de mensagens em um mesmo contexto, permite o envio de notificações simultaneamente a um grande número de assinantes [23].

Estes serviços podem ser utilizados em casos onde um ou mais usuários precisem ser notificados sobre algum acontecimento. Para tanto, basta que o sistema utilize uma API, já que todo o fluxo de negociação com sistemas de *push notifications*, provedores de email e redes telefônicas fica a cargo da plataforma de nuvem.

3.1.5 Integração

Na figura 7 temos a arquitetura de nuvem desenvolvida para suportar a integração entre aplicações móveis e dispositivos IoT. Separando funcionalmente os componentes discutidos nós temos as estruturas de acesso, armazenamento, computação e notificação.

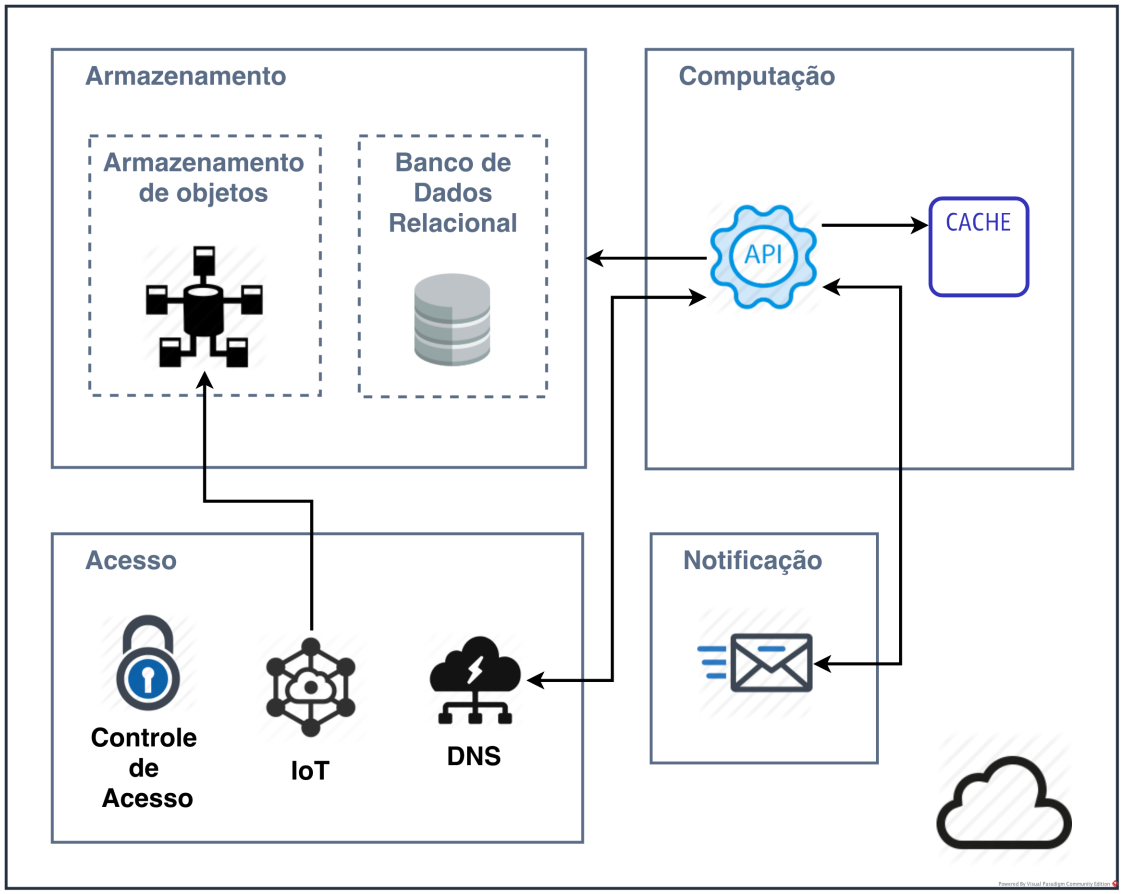
A nuvem poderá ser acessada a partir de dois pontos, o sistema de integração IoT e o DNS de nuvem. Todos os dispositivos IoT utilizados pelo sistema se conectarão ao serviço de integração utilizando algum dos protocolos suportados. As requisições REST dos dispositivos móveis serão roteadas para acessar a API através do DNS. Ambos pontos de acesso utilizam o sistema de controle de acesso para proteger seus recursos e serviços.

O armazenamento de dados brutos gerados pelos dispositivos IoT será feito pelo serviço de armazenamento de objetos, já a aplicação utilizará o serviço de armazenamento de dados estruturados em banco de dados. O armazenamento de objetos pode também ser utilizado para a criação de *data lakes*, *data warehouses* e *data marts* devido a sua capacidade de escalabilidade automática e suas ferramentas de busca e tratamento de dados. Os administradores do sistema podem definir regras para a migração automática de dados antigos, ou de ex-clientes, para a classe de arquivamento a fim de diminuir o custo com armazenamento.

O serviço de computação hospedará o servidor da aplicação utilizando *containers* Docker, para maior performance é interessante que o sistema operacional da máquina seja Linux para utilizar a integração nativa com o motor do Docker. Todo software necessário para o funcionamento da aplicação estará nos containers, incluindo servidores *web* e de *cache*, microsserviços e a API que será discutida em detalhes na Seção 3.2.

Por fim, o sistema de notificação realiza o envio de alertas aos usuários por meio de *push notifications*, *emails* e mensagens de celular. Este serviço também pode ser utilizado para integrar microsserviços ao sistema, como por exemplo um processador de dados, muitas vezes necessário devido ao grande volume de dados que dispositivos IoT podem gerar.

Figura 7 – Diagrama de arquitetura da nuvem



Fonte: o autor.

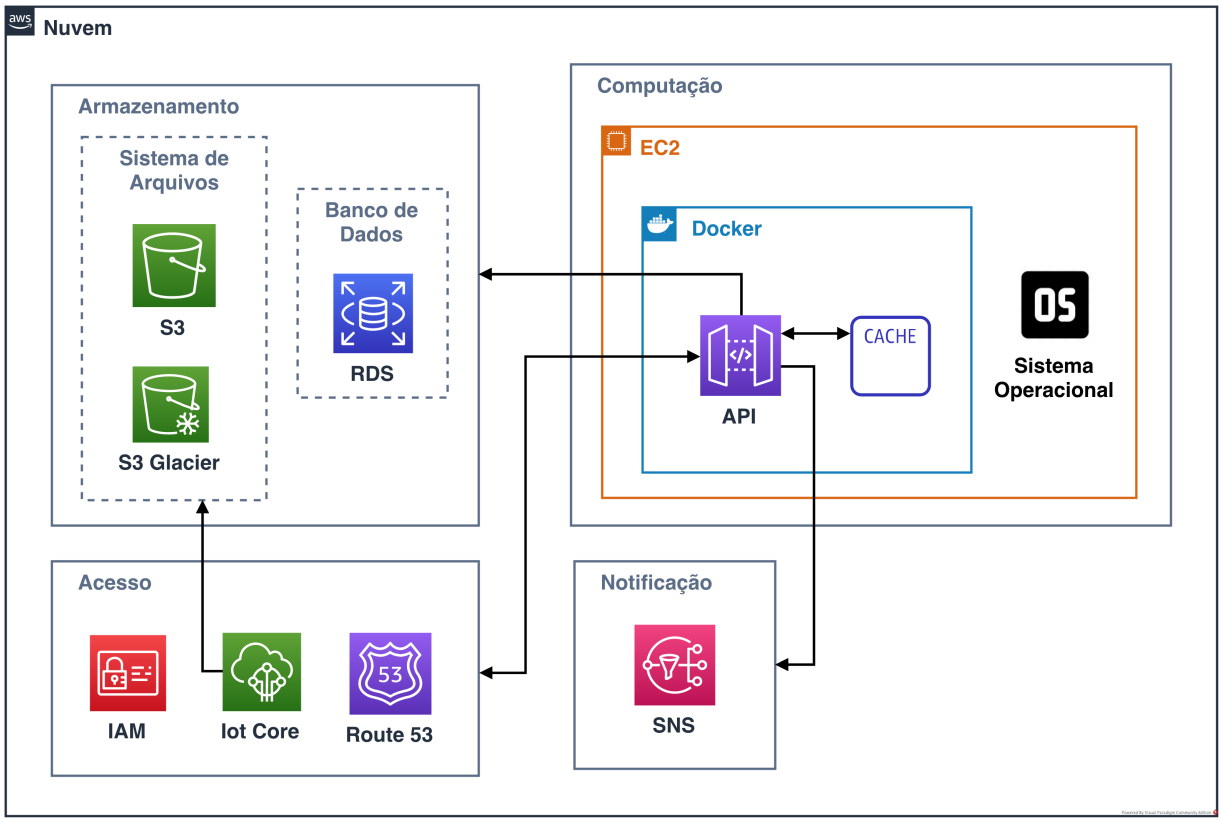
3.1.6 Implantação

Para exemplificar os conceitos da arquitetura de nuvem definida nesta seção, foi escolhida como plataforma a nuvem da AWS. A implantação de serviços na AWS é feita completamente através do *AWS Console*, uma plataforma *online* onde é possível criar um ambiente na nuvem e alocar, gerenciar e monitorar serviços. A partir da arquitetura definida na figura 8 podemos realizar uma implantação inicial alocando os serviços da seguinte forma:

- O serviço de computação **EC2** utilizando uma máquina *t2.medium*, que possui 2 processadores, 4GB de memória RAM e 8GB de armazenamento SSD, funcionando 24 horas por dia.
- O armazenamento de objetos **S3** com capacidade de armazenar 100GB de dados por mês e realizando 200.000 acessos para criação e leitura de dados.

- O DNS em nuvem **Route 53** com um domínio cadastrado e 10 milhões de acessos mensais.
- O banco de dados relacional **RDS** utilizando uma máquina *t3.small*, que possui 2 processadores e 2GB de RAM, funcionando 24 horas por dia.
- O serviço de notificação **SNS**. Ele fornece 1 milhão de notificações grátis mensais e cobra $0.50 a partir do segundo milhão de mensagens no mês.
- O armazenamento de objetos de longo prazo **Glacier** com armazenamento de 2GB mensais de dados. Existe um custo para acesso aos dados por GB transferido.
- O serviço de integração **IoT Core** para 10.000 dispositivos IoT enviando mensagens a cada 5 minutos e que podem receber comandos nesta mesma frequência.
- **Tráfego de dados** padrão, onde somente o tráfego saindo da nuvem, *output*, é cobrado.

Figura 8 – Diagrama de arquitetura da nuvem AWS



Fonte: o autor.

3.1.7 Custo

A nuvem da AWS utiliza um modelo de cobrança sob demanda onde serviços são cobrados à medida que são utilizados. A tabela 1 mostra uma estimativa de custo mensal para a implantação da arquitetura com as configurações discutidas quando hospedados na região *US East (North Virginia)*. Os valores foram obtidos utilizando a calculadora, *Simple Monthly Calculator*, da AWS.

Tabela 1 – Projeção de custo mensal da nuvem AWS

Serviço	**Custo mensal (Dólar)**
EC2	34.77
S3	2.32
RDS	27.19
Glacier	0.01
Iot Core	90.00
Route 53	4.50
SNS	0.00
Trafego de dados	1.35
Total	**160.14**

3.2 *Back-end*

O *Back-end* do sistema é o componente que está entre a interface de usuário e o banco de dados. Sua função é tratar as requisições enviadas pelo usuário, consultar ou modificar o banco de dados e devolver uma resposta à interface. No caso de sistemas de gerenciamento, o *back-end* é um *web service* no qual a interface de usuário acessa através de uma API para realizar operações.

É interessante implementar o *back-end* utilizando linguagens de programação com bom suporte a operações de comunicação já que o acesso a nossa API será através da *internet*. Linguagens com boas ferramentas para a criação de *web services* incluem Python, Javascript, PHP e C#, mas qualquer linguagem com suporte a requisições HTTP é suficiente.

3.2.1 *RESTful API*

Uma API é dita *RESTful* quando a mesma adere ao padrão REST discutido na Seção 2.4.3. A utilização deste padrão pelo servidor proporciona uma integração consistente com a interface de usuário além de permitir a reutilização de código e mantenabilidade.

O acesso à API é realizado através de requisições HTTP que são compostas por um URI, comumente chamado de rota, para identificação de recursos, um método HTTP e opcionalmente um corpo da requisição onde pode ser enviada informações ao servidor. A resposta a uma requisição é composta por um código de estado HTTP e o corpo da resposta que trás os dados requisitados ou mensagens de sucesso ou erro.

Os métodos HTTP mais comuns são GET, POST, PUT e DELETE para a busca, criação, modificação e remoção de recursos da aplicação. A API precisa reconhecer as cinco classes de códigos de estado HTTP: 1xx para informações, 2xx para sucesso, 3xx para redirecionamento, 4xx para erro no cliente e 5xx para outros erros [24].

O corpo das requisições e respostas deve ter um formato conhecido pelo cliente e o servidor, este formato é definido pelo tipo de mídia da comunicação. Os tipos são definidos em [25] e cobrem de simples mensagens de texto a programas executáveis. O tipo `application/json` é o mais utilizado em *web services* já que o JSON, *Javascript Object Notation*, é um formato de dado bastante simples que pode representar quatro tipos primitivos (*strings*, números, booleanos e *null*) e dois tipos de estruturas (objetos e *arrays*) [26].

Um exemplo de operação da API é a busca por um recurso específico. A requisição `GET /api/usuarios/27` utiliza o método `GET`, portanto está buscando um recurso, e a rota `/api/usuarios/27` indica que o recurso em questão é um usuário do sistema com o identificador 27. Em caso de sucesso, o resultado desta requisição deve ter código 200 e um corpo num formato semelhante ao da figura 9.

Figura 9 – Exemplo de resposta do tipo *application/json*

```
{
   "data": {
       "id": 27,
       "nome": "João Silva",
       "cpf": "123.456.789-90",
       "criado_em": "2019-11-10 10:32:21"
   }
}
```

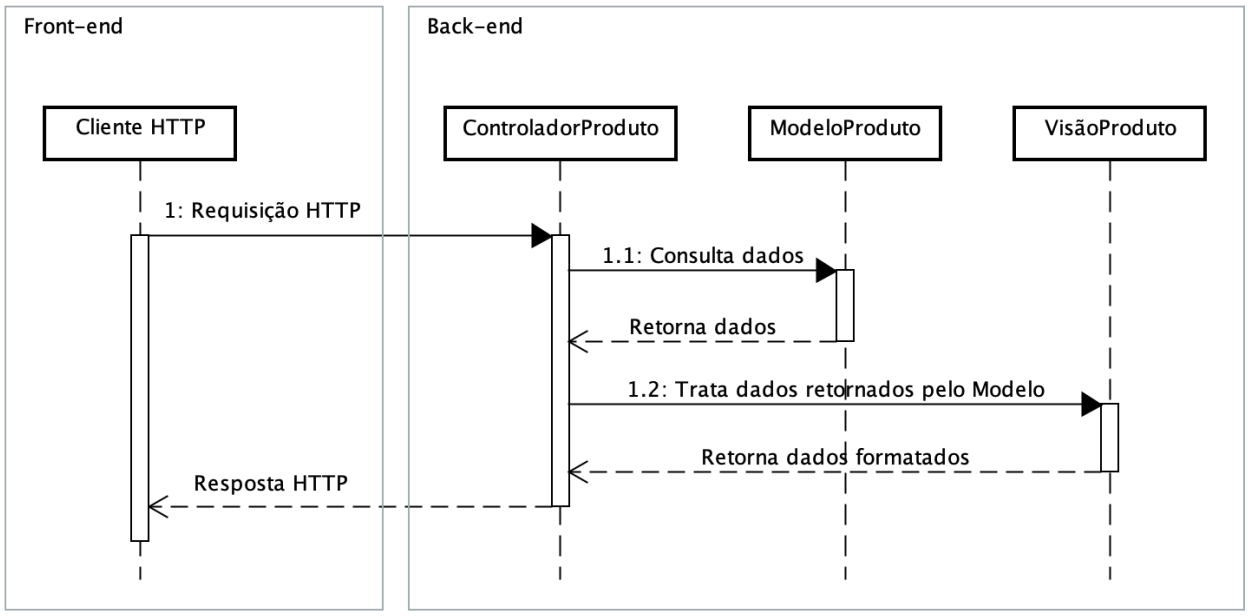
Fonte: o autor.

3.2.2 Estrutura

A arquitetura MVC, discutida no item 2.4.2, foi utilizada para a criação do *back-end*. Cada rota da API deve utilizar um controlador, responsável por interpretar a requisição HTTP e requisitar dados ao modelo. O modelo possui a lógica de negócio, as representações de cada entidade do banco de dados para realizar consultas. A visão neste contexto será o componente de *software* responsável por produzir respostas HTTP no formato esperado pela aplicação móvel.

A figura 10 mostra um exemplo da operação da API ao ser utilizada pela aplicação móvel para buscar produtos. A aplicação realiza uma requisição GET à API e é roteada para o controlador de produto. O controlador requisita os dados ao modelo de produto que por sua vez faz uma consulta ao banco de dados e retorna os dados dos produtos buscados. O controlador então encaminha os dados para a visão de produto que trata estes dados e os retorna para o controlador que irá montar e enviar a resposta HTTP para a aplicação.

Figura 10 – Diagrama de sequência de uma API implementada com o padrão MVC.



Fonte: o autor.

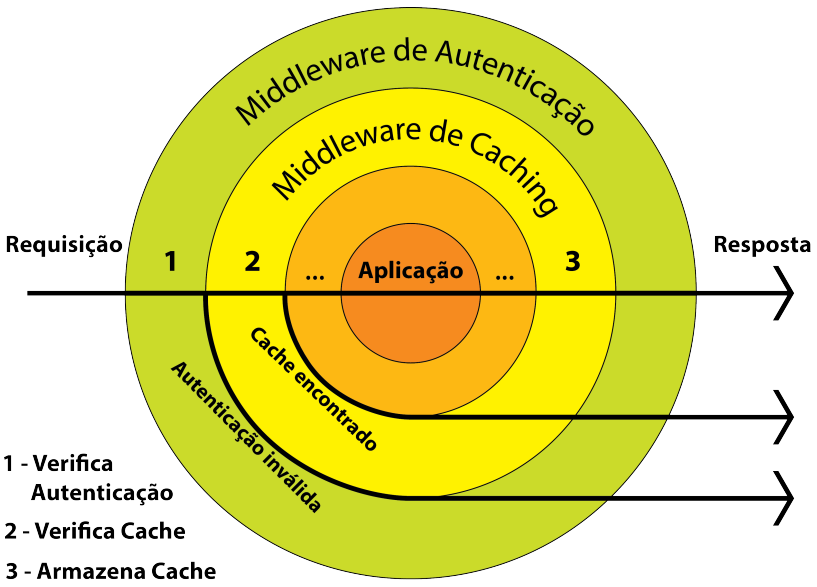
Como a interação com este tipo de API é estritamente através de requisições HTTP, ou seja, sem uma interface gráfica, é comum a união da visão com o controlador. Desta forma, o início da operação se dá no recebimento da requisição pelo controlador correspondente, que ao final do processo também produz a resposta HTTP.

3.2.3 *Middlewares*

A estruturação do sistema em camadas permite a utilização de um componente de *software* chamado de *middleware*, funções que são executadas entre operações para modificar valores de entrada ou saída. Desta forma podemos aumentar a performance e mantenabilidade do sistema ao desacoplar a lógica de negócios de códigos de validação de entrada, autenticação de usuários, *Caching* de dados entre outros.

A figura 11 mostra o fluxo de dados numa API *RESTful* que utiliza uma estrutura em camadas. A aplicação gerencia uma pilha de *middlewares* de acordo com sua ordem de execução, os três parâmetros que cada uma destas funções recebem são os objetos de requisição e resposta e uma função `next()`, que invoca o próximo *middleware* da pilha. Os objetos de requisição e resposta podem ser modificados livremente de acordo com a finalidade do *middleware*, que em casos como o de autenticação inválida pode finalizar o ciclo de requisição-resposta.

Figura 11 – API estruturada em camadas



Fonte: adaptado de (Boronczyk, 2013).

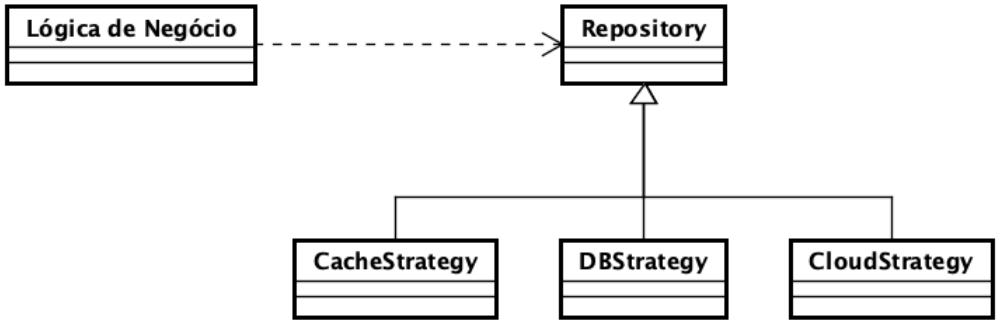
3.2.4 Acesso aos dados

Em uma aplicação que envolve sistemas IoT o acesso aos dados da aplicação normalmente é realizado de duas maneiras. Dados gerais da aplicação são armazenados em um banco de dados relacional e os dados provenientes dos dispositivos IoT são armazenado em um armazenado de objetos em nuvem.

Desta forma é interessante que o Modelo utilize um mecanismo para coordenar o acesso aos dados nestes locais. O padrão ***Repository*** faz a mediação entre o Modelo e a camada de mapeamento de dados atuando como uma coleção de objetos do domínio. Repositórios fornecem uma visão orientada a objetos da camada de persistência ao encapsular os objetos persistidos no armazenamento e suas operações [28]. Este padrão centraliza o acesso aos dados favorecendo a mantenabilidade da aplicação já que mudanças nas estruturas de armazenamento causam reescrita de código apenas no repositório.

A figura 12 trás uma visão geral do padrão *Repository* demonstrando a separação de formas de acesso a diferentes estruturas de armazenamento de dados, como por exemplo *cache*, banco de dados relacional e armazenamento em nuvem.

Figura 12 – Diagrama UML do Modelo utilizando o padrão Repository



Fonte: o autor.

Em conjunto com os repositórios é possível utilizar um ORM, do inglês *object-relational mapping*, para tornar as operações com as entidades do Modelo orientada a objetos. ORMs realizam um mapeamento entre os objetos definidos no *back-end* com as entidades que eles representam no banco de dados.

A figura 13 representa um processo de consulta ao banco de dados utilizando repositórios com ORM, tornando a interação com os dados mais prática e fácil de entender. O repositório fornece uma forma de consulta que abstrai a implementação das estruturas de acesso aos dados e o ORM representa os dados através de objetos que possuem os atributos e relacionamentos das entidades do banco de dados.

3.2.5 *Caching* de dados

A quantidade de dados gerada por dispositivos IoT pode trazer um grande impacto à performance da aplicação caso seja necessário realizar computações sobre estes dados a cada requisição enviada pelos usuários. Uma forma de mitigar este problema é a utilização de um servidor de *cache* para armazenar o resultado mais recente de cada rota a medida que elas são chamadas.

Figura 13 – Exemplo de consulta de dados utilizando o padrão *Repository* e ORM

```
let id_produto   = 10;
let produto      = produtoRepository.getProduto(id_produto);
let nome_produto = produto.getNome();
```

Fonte: o autor.

Servidores de *cache* normalmente utilizam uma organização de chave-valor para o armazenamento dos dados, muitas vezes na memória da sua máquina para diminuir o tempo de resposta do sistema. É necessário um mecanismo de sincronização para garantir a validade dos dados, normalmente este mecanismo é acionado pelo *back-end* a cada vez que dados relevantes são modificados ou adicionados às estruturas de armazenamento.

A utilização deste sistema segue o fluxo mostrado na figura 11, um *middleware* de *caching* verifica se existe uma resposta armazenada no servidor de *cache* para cada requisição recebida, em caso positivo esta resposta é imediatamente enviada ao usuário sem que seja necessário reprocessar a requisição, caso contrário a requisição é processada e a resposta é armazenada para futuras consultas.

3.3 *Front-end*

O *Front-end* do sistema é a interface que o usuário final utliza, neste caso uma aplicação móvel disponível para as plataformas iOS da *Apple* e Android do *Google*. Para atender o requisito de mantenabilidade, deve ser utilizado um *framework* de aplicações multiplataforma, discutido na Seção 2.3.3.

3.3.1 Componentes

Uma interface de usuário é desenvolvida a partir da combinação de componentes, funções que normalmente retornam trechos de código em linguagem de marcação. Sua estruturação é parecida com o HTML usado em *web sites*, mas utilizando *tags* próprias, muitas vezes criadas pelos desenvolvedores.

Um componente é uma função que recebe um parâmetro com suas propriedades, possui um estado próprio e retorna um código em linguagem de marcação. A maioria dos componentes podem ser customizados no momento de sua criação através dos parâmetros de criação, desta forma é possível reutilizá-los facilmente ao longo da aplicação.

O estado é um conjunto variáveis que podem ser alteradas durante a execução da aplicação modificando funcionalidades e o estilo visual de um componente, podendo ser definido localmente em um componente ou utilizando o conceito de containers de estado que será discutido na Seção 3.3.2.

A estilização de aspectos como posicionamento, cores e fontes dos componentes pode ser feita através de uma propriedade chamada `style`, utilizando objetos *Javascript* que utilizam a mesma nomenclatura e conceitos do CSS, cascading style sheet, utilizado em *web sites*.

A figura 14 mostra um exemplo de componente em *React Native* que define um cabeçalho para a página, sendo criado através do componente nativo `<Text>` e que pode ser utilizado em outros componentes como `<Titulo texto="Página Inicial"/>`.

Figura 14 – Exemplo de componente em *React Native*

```
function Titulo(props) {
    const texto = props.texto;
    return (
        <Text style={{fontSize: 32, padding: 10, fontWeight: 'bold'}}>
            {texto}
        </Text>
    );
}
```

Fonte: o autor.

Bibliotecas externas, como o Native-Base, fornecem códigos para a criação de todos os componentes nativos de ambas plataformas. Desta forma é possível deixar o desenvolvimento mais eficiente ao focar apenas na visualização e funcionalidade dos componentes ao invés de sua implementação.

3.3.2 Padrão *Flux*

A natureza de sistemas de gerenciamento em exibir varias informações simultaneamente trás a necessidade à aplicação de coordenar o fluxo de dados entre seus diversos componentes. Desta forma é possível a utilização do padrão Flux, uma variação do padrão *Observer*, que centraliza o estado da aplicação e garante um fluxo de dados unidirecional para permitir a escalabilidade de interfaces de usuário. O padrão Flux pode ser descrito a partir de três princípios:

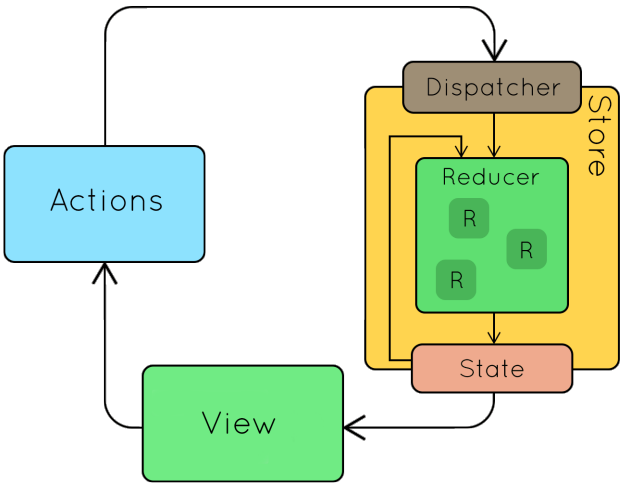
1. Fonte única de verdade: O estado da aplicação é armazenado numa árvore de objetos dentro de um único local chamado de *store*.

2. O Estado é somente-leitura: A única maneira de modificar o estado é emitindo uma ação, um objeto que descreve a mudança que deve acontecer.

3. Mudanças são realizadas por funções determinísticas: Funções de redução, *reducers*, são utilizadas para especificar como a árvore de estados é transformada por ações.

A figura 15 é uma visão destes princípios e do fluxo de dados no padrão Flux. Um componente da interface do usuário, *View*, envia uma ação, *Action*, ao realizar alguma operação. O expedidor, *Dispatcher*, recebe este objeto e o encaminha para o *Reducer* adequado, no qual a partir do estado atual do componente e do conteúdo presente na ação produz um novo estado. Este novo estado é armazenado no *Store* e encaminhado para o componente para que possa ser exibido ao usuário.

Figura 15 – Diagrama do padrão *Flux*.

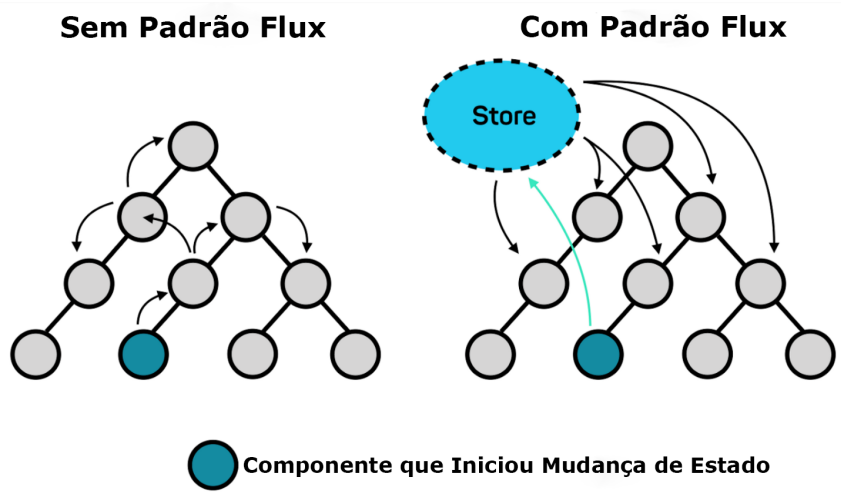


Fonte: adaptado de (Terpil, 2015).

Com este padrão é possível compartilhar o estado de um componente automaticamente quando uma mudança ocorrer, mantendo a coerência entre a visualização e os dados. A figura 16 mostra o processo de propagação do estado de um componente a outros componentes que estão interessados.

No caso onde o padrão Flux não é utilizado, o componente que teve o estado alterado precisa propagar esta alteração manualmente a cada componente interessado. Quando o Flux é utilizado, no momento em que o estado do componente é alterado no *Store* todos os outros componentes interessados são notificados automaticamente da mudança e recebem o novo estado.

Figura 16 – Propagação de estado com e sem Flux.



Fonte: adaptado de (Weck, 2019).

3.3.3 Armazenamento

Aplicações podem armazenar dados na memória interna do dispositivo móvel para diminuir a carga de requisições ao *back-end* ou manter as configurações do usuário, por exemplo. Esta funcionalidade deve ser utilizada com cuidado, componentes críticos da aplicação não devem ser totalmente dependentes do armazenamento interno já que erros por falta de espaço de espaço de armazenamento ou permissões de acesso podem ser comuns.

As formas mais comuns para persistir dados da aplicação são através de bancos de dados locais como SQLite e armazenamento nativo dos dispositivos chamado de *LocalStorage*. Ambas tecnologias são recursos exclusivos de cada aplicação não sendo possível o compartilhamento de dados entre diferentes aplicações.

Implementações do padrão Flux oferecem integração a estes sistemas, facilitando ainda mais o manuseio dos dados presentes na aplicação. É comum armazenar as credenciais dos usuários em uma seção segura do *LocalStorage*, dessa forma o usuário não necessita realizar o login cada vez que ele retorne ao aplicativo. Para tanto, o Store é extendido virtualmente para ter acesso aos armazenamentos locais e manter os dados acessíveis usando sua interface padrão.

3.3.4 Cliente HTTP

O acesso ao servidor é realizado através de um cliente que se conecta à API através de requisições HTTP seguindo o padrão REST. Cada *framework* fornece uma forma de acesso ao cliente nativo, tornando possível realizar vários tipos de interações HTTP.

Bibliotecas externas podem expandir as funcionalidades do cliente padrão ao utilizar funções específicas a cada método HTTP como get(), post(), put(), delete() , configurações globais e requisições concorrentes.

A figura 17 mostra um exemplo do código necessário para realizar uma requisição à API em *Javascript*. O cliente HTTP pode ser configurado como um serviço, desta forma ele é acessível em todas as partes da aplicação e concentra sua configuração, como endereço da API, cabeçalhos e cookies, em apenas um arquivo para permitir que seja facilmente mantido.

Figura 17 – Cliente HTTP realizando requisição de produtos à API.

```
const resposta  = await client.get("https://www.empresa.com.br/api/produtos");
let produtos    = resposta.data;
```

Fonte: o autor.

4 Resultados

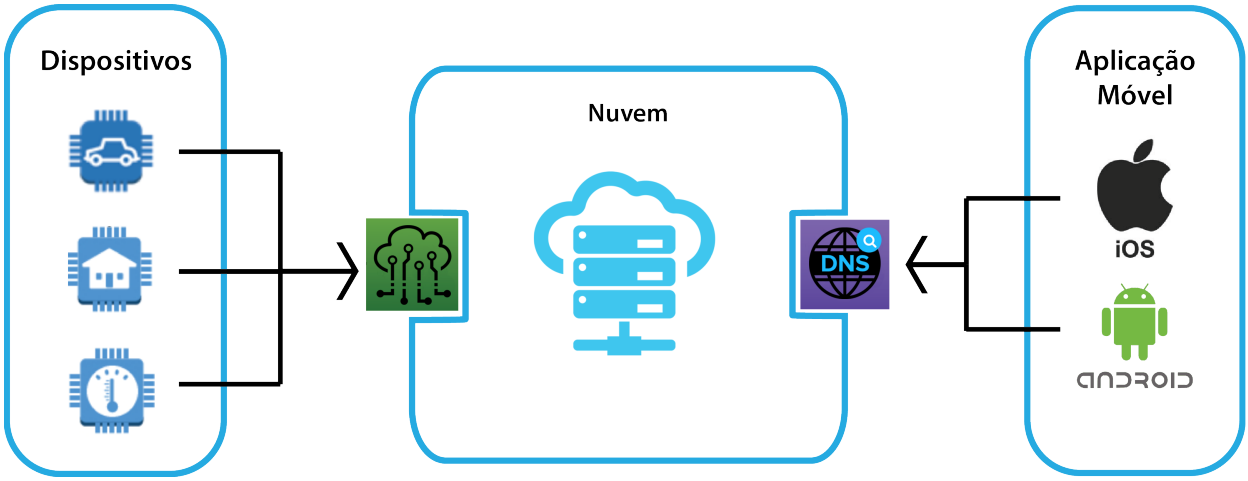
Este Capítulo exibe o resultado da integração entre os componentes da arquitetura produzida e sua aplicação em um sistema real. Na Seção 4.1 os componentes da arquitetura são integrados produzindo um novo modelo visual. Por fim, na Seção 4.2 será discutida uma prova de conceito desta arquitetura que está sendo investigada.

4.1 Modelo

Obtivemos uma arquitetura de *software* para desenvolvimento de soluções que englobam sistemas IoT, servidores para tratamento e análise de dados e interfaces de usuário através de aplicações móveis, tudo isso levando em consideração requisitos de escalabilidade, mantenabilidade, confiabilidade e a agilidade no desenvolvimento do sistema.

O diagrama da figura 18 mostra a integração entre os três componentes da arquitetura discutidos no Capítulo 3. A nuvem especificada na Seção 3.1 integra os dispositivos IoT à aplicação móvel, sua arquitetura pode ser vista na figura 7. O ponto de acesso dos dispositivos é a partir do serviço de integração, toda a conexão e troca de mensagens é coordenada por esse serviço. Já as aplicações móveis acessarão o sistema a partir do DNS ao utilizarem alguma chamada REST para um domínio controlado por este serviço.

Figura 18 – Diagrama da arquitetura



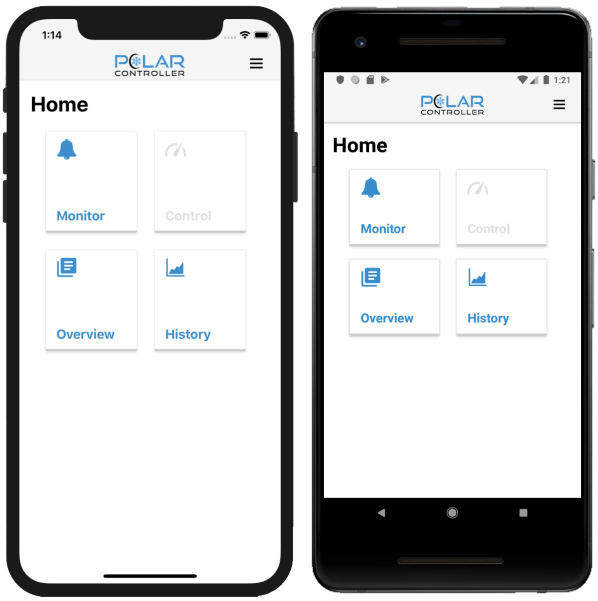
Fonte: o autor.

4.2 Prova de Conceito

Uma prova de conceito está sendo desenvolvida para uma empresa dos Estados Unidos utilizando os componentes aqui apresentados. O sistema envolve dispositivos IoT conectados a aparelhos de refrigeração como ar-condicionados centrais e freezers industriais. Estes dispositivos coletam dados dos aparelhos e do ambiente, a partir de inúmeros sensores, e enviam pacotes de informação a cada 7 minutos ao servidor da aplicação para que sejam processados e tenham o resultado armazenado num banco de dados relacional.

A partir deste dado estruturado, uma aplicação móvel multiplataforma realiza a exibição das informações do sistema de refrigeração do cliente. Desta forma, atualmente é possível acompanhar aspectos como consumo de energia, variação de temperatura e o funcionamento de cada componente dos aparelhos, onde em caso de falhas o cliente é alertado através de notificações. A figura 19 mostra a aplicação funcionando nas plataformas iOS e Android.

Figura 19 – Aplicação nas plataformas iOS e Android, respectivamente



Fonte: o autor.

4.2.1 Estrutura

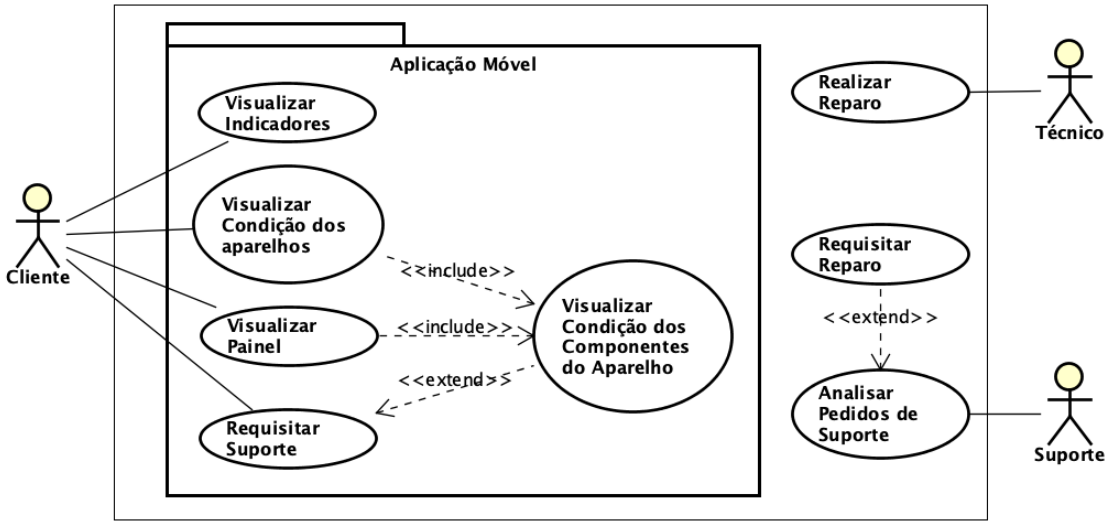
Neste momento já está em estágio de desenvolvimento a infraestrutura do sistema utilizando a nuvem da AWS como IaaS. O *back-end* do sistema utiliza a plataforma de computação EC2, um banco de dados MySQL no serviço RDS e por fim, os dispositivos IoT enviam seus dados para um armazenamento de objetos S3.

A partir de um *back-end* construído em Python3 utilizando o *framework* Django, uma aplicação para as plataformas iOS e Android escrita em *React Native* acessa dados de dispositivos IoT e exibe as diversas informações obtidas a partir da análise destes dados.

Existe a previsão da implementação de um *data mart* para permitir a escalabilidade do sistema sem comprometer sua performance. Além disso, os dados disponíveis nessa estrutura permitirão a utilização de inteligência computacional para o controle automático dos aparelhos de refrigeração a partir dos dispositivos IoT.

A figura 20 mostra os principais casos de uso de cada ator do sistema. A aplicação móvel é acessível somente aos Clientes, no qual podem visualizar indicadores do sistema; visualizar a condição de seus aparelhos; visualizar a condição das partes de cada aparelho; visualizar o painel do sistema e requisitar suporte. Os atores Suporte e Técnico fazem parte da empresa e permitem o correto funcionamento dos sistemas do Cliente.

Figura 20 – Diagramas de casos de uso do sistema



Fonte: o autor.

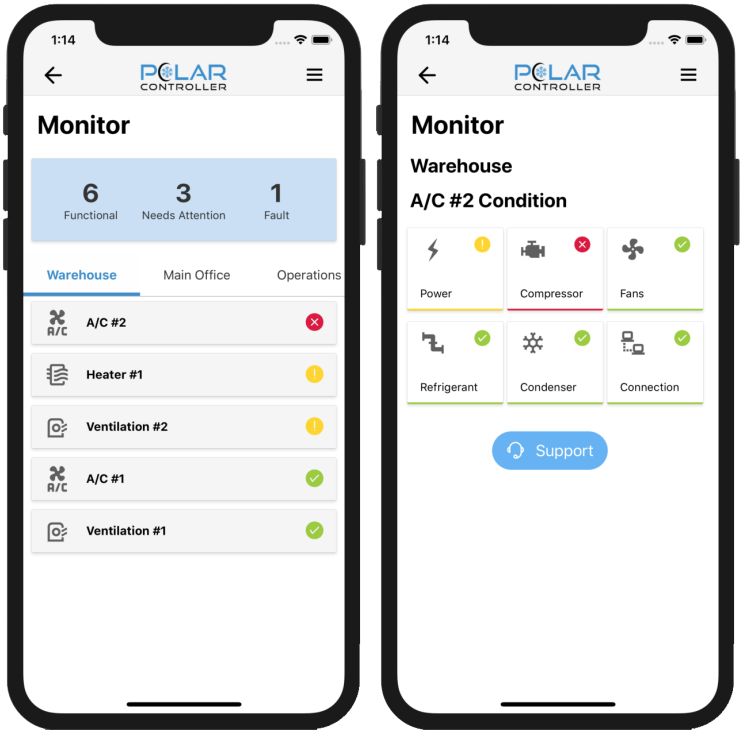
4.2.2 Aplicação Multiplataforma

Os aparelhos dos clientes são conectados a dispositivos IoT que coletam informações sobre seu funcionamento e condições ambientais. A aplicação permite que clientes possam visualizar de forma intuitiva estas informações e agir em caso de defeito. Os conceitos apresentados na Seção 3.3 foram utilizados para o desenvolvimento da aplicação a fim de permitir uma maior agilidade na entrega de versões para as plataformas iOS e Android.

Os componentes utilizados na interface de usuário são nativos às plataformas para tornar sua utilização intuitiva e diminuir o tempo de aprendizado necessário para o uso da aplicação.

A figura 21 mostra os casos de uso de visualização da condição dos aparelhos e de seus componentes, respectivamente. A tela *Monitor* mostra ao cliente todos os seus dispositivos organizados em abas e listas onde é possível ver a condição do aparelho através de ícones coloridos que representam alertas de estados de funcionamento. Ao selecionar um aparelho o cliente pode visualizar a condição de cada parte do aparelho e, se necessário, pode requisitar suporte técnico à empresa.

Figura 21 – Telas de condição dos aparelhos e componentes

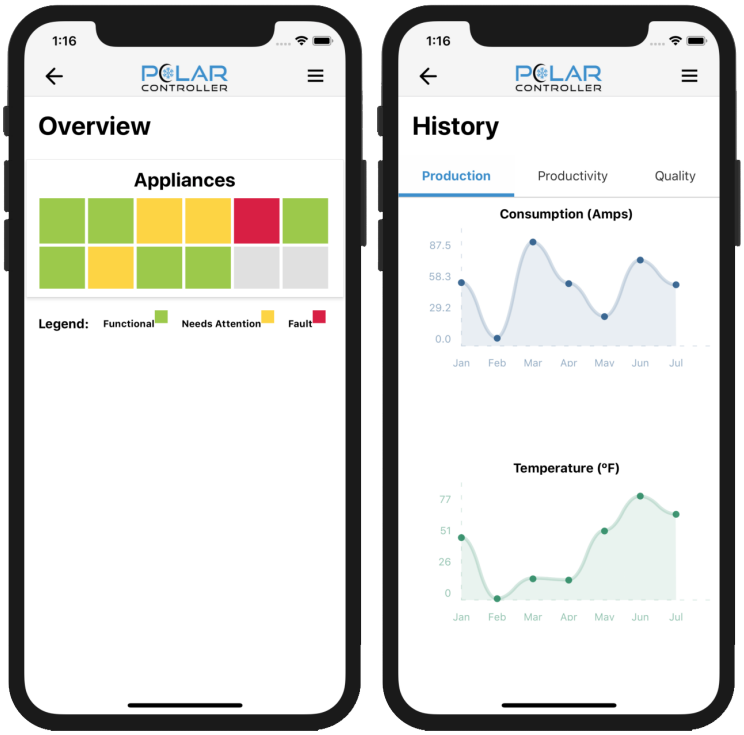


Fonte: o autor.

A figura 22 mostra as telas de visão geral do sistema. O painel, *Overview*, é uma visualização que simplifica o acesso a informações sobre a condição de todos os aparelhos monitorados do cliente. A cor de cada célula da tabela representa o estado de um aparelho, utilizando a mesma escala de cores de estados utilizado ao longo da aplicação para manter coerência entre as visualizações.

A tela *History* exibe indicadores de produção, produtividade e qualidade de todo o sistema do cliente. Desta forma o cliente pode acompanhar o funcionamento e a evolução de seus equipamentos em relação a performance, economia entre outros aspectos. Esta visualização também será importante quando o sistema permitir o controle dos aparelhos, assim o cliente pode perceber mais facilmente se as mudanças realizadas estão surtindo efeito em seus aparelhos.

Figura 22 – Telas de painel de aparelhos e de indicadores do sistema



Fonte: o autor.

5 Conclusões e Trabalhos Futuros

Neste trabalho foi proposta uma arquitetura de *software* para integração de sistemas de gerenciamento IoT a aplicações móveis multiplataforma. Esta implementação poderá ser amplamente utilizada por empresas que fabricam *softwares* atualmente e que não contemplam demandas de IoT para monitoramento, o que levará a uma maior agilidade e assertividade nas suas especificações e desenvolvimentos, possivelmente ajudando a aumentar a adoção das tecnologias da 4ª Revolução Industrial.

O aumento na disponibilidade de dispositivos IoT e na qualidade das infraestruturas de redes são uma grande oportunidade para o desenvolvimento de sistemas de gerenciamento para otimizar processos industriais e de aspecto social como, por exemplo, mobilidade urbana e saúde. Espera-se que os artefatos produzidos na monografia possam embasar o desenvolvimento de softwares multiplataforma com integração a tais sistemas IoT.

Durante o desenvolvimento da prova de conceito diversas novas tecnologias foram conhecidas, permitindo novas visões sobre o problema. Algumas delas foram integradas a arquitetura aqui proposta, outras que ficaram de fora podem oferecer grandes vantagens aos sistemas que serão implementados e serão comentadas agora.

A utilização de outros serviços disponíveis na nuvem AWS pode aumentar a performance do sistema em diversas frentes. Para o manuseio dos dados brutos seria interessante a aplicação de *Data Lakes* e *Data Marts*, oferecido pelos serviços *Lake Formation* e *Redshift*. Desta forma a carga de processamento na API seria reduzida já que dados tratados estariam sempre disponíveis, além de permitir a aplicação de técnicas de aprendizado de máquina.

Por fim, sistemas de gerenciamento podem ser aprimorados com a utilização de algoritmos de inteligência artificial para predição de eventos, permitindo atuações preventivas e corretivas sobre os dispositivos IoT. A AWS também fornece serviços de inteligência computacional que podem facilitar este processo, incluindo o *SageMaker*. Devido a modularização desta arquitetura, novos serviços podem ser adicionados sem muito esforço, possivelmente aumentando o impacto das soluções que as utilize.

Referências

1 ITU-T. *Overview of the Internet of things*. [S.l.], 2012. Disponível em: <http://handle.itu.int/11.1002/1000/11559>. Acesso em: 25 de setembro de 2019. Citado 4 vezes nas páginas 14, 16, 17 e 18.

2 FORBES. *2018 Roundup Of Internet Of Things Forecasts And Market Estimates*. [S.l.], 2018. Disponível em: <https://www.forbes.com/sites/louiscolumbus/2018/12/13/2018-roundup-of-internet-of-things-forecasts-and-market-estimates>. Acesso em: 20 de setembro de 2019. Citado na página 14.

3 FORBES. *Mobile App 'State of Mobile 2019 Report' From App Annie*. [S.l.], 2019. Disponível em: <https://www.forbes.com/sites/tjmccue/2019/01/30/mobile-app-state-of-mobile-2019-report-from-app-annie>. Acesso em: 21 de setembro de 2019. Citado 2 vezes nas páginas 15 e 22.

4 LATIF, M. et al. Cross platform approach for mobile application development: A survey. *International Conference on Information Technology for Organizations Development (IT4OD)*, February 2016. Citado na página 15.

5 IEEE. *Internet of Things (IoT) Ecosystem Study*. [S.l.], 2015. Disponível em: <https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/other/iot_ecosystem_exec_summary.pdf>. Acesso em: 17 de dezembro de 2019. Citado na página 16.

6 MELL, P.; GRANCE, T. *The NIST Definition of Cloud Computing*. [S.l.], 2011. Citado 2 vezes nas páginas 19 e 20.

7 SOLO NETWORK. *Diferenças entre IaaS, SaaS e PaaS*. [S.l.], 2019. Disponível em: <https://www.solonetwork.com.br/Produtos/Microsoft/microsoft-azure>. Acesso em: 27 de outubro de 2019. Citado na página 21.

8 POSLAD, S. *Ubiquitous Computing: Smart Devices, Environments and Interactions*. 1. ed. [S.l.]: John Wiley & Sons, 2009. Citado na página 22.

9 STATCOUNTER. *Mobile Operating System Market Share Worldwide - October 2019*. [S.l.], 2019. Disponível em: <https://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 9 de novembro de 2019. Citado na página 22.

10 GUERRA, Q. F. *Aplicação em diferentes plataformas*. [S.l.], 2015. Disponível em: <https://market.ionicframework.com/starters/multiplatform>. Acesso em: 9 de novembro de 2019. Citado na página 24.

11 PRESSMAN, R. S. *Engenharia de Software - Uma Abordagem Profissional*. 7. ed. [S.l.]: McGrawHill, 2011. Citado na página 23.

12 BUSCHMANN, F. et al. *Pattern Oriented Software Architecture: A system of patterns*. [S.l.]: John Wiley & Sons, 2001. v. 1. (Software Design Patterns, v. 1). Citado na página 24.

13 W3C. *Web Services Architecture*. [S.l.], 2004. Disponível em: <https://www.w3.org/TR/ws-arch/>. Acesso em: 03 de novembro de 2019. Citado na página 26.

14 FIELDING, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. Tese (Doutorado) — University of California, Irvine, 2000. Citado na página 26.

15 AMAZON WEB SERVICES. *Route 53*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/route53/>. Acesso em: 10 de novembro de 2019. Citado na página 27.

16 AMAZON WEB SERVICES. *Identity and Access Management*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/iam/>. Acesso em: 10 de novembro de 2019. Citado na página 28.

17 AMAZON WEB SERVICES. *IoT Core*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/iot-core/>. Acesso em: 12 de novembro de 2019. Citado na página 28.

18 MESNIER, M.; GANGER, G.; RIEDEL, E. Object-based storage. *IEEE Communications Magazine*, v. 41, agosto 2003. Citado na página 28.

19 AMAZON WEB SERVICES. *Simple Storage Service*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/s3/>. Acesso em: 10 de novembro de 2019. Citado na página 29.

20 AMAZON WEB SERVICES. *Relational Database Service*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/rds/>. Acesso em: 10 de novembro de 2019. Citado na página 29.

21 AMAZON WEB SERVICES. *Elastic Compute Cloud*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/ec2/>. Acesso em: 10 de novembro de 2019. Citado na página 29.

22 DOCKER. *What is a container?* [S.l.], 2019. Disponível em: <https://www.docker.com/resources/what-container>. Acesso em: 10 de novembro de 2019. Citado na página 30.

23 AMAZON WEB SERVICES. *Simple Notification Service*. [S.l.], 2019. Disponível em: <https://aws.amazon.com/sns/>. Acesso em: 10 de novembro de 2019. Citado na página 31.

24 FIELDING, R.; RESCHKE, J. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. [S.l.], 2014. Disponível em: <https://tools.ietf.org/html/rfc7231>. Acesso em: 15 de novembro de 2019. Citado na página 35.

25 FREED, N.; KUCHERAWY, M. *Media Types*. [S.l.], 2019. Disponível em: <https://www.iana.org/assignments/media-types/media-types.xhtml>. Acesso em: 15 de novembro de 2019. Citado na página 35.

26 BRAY, T. *The JavaScript Object Notation (JSON) Data Interchange Format*. [S.l.], 2017. Disponível em: <https://tools.ietf.org/html/rfc8259>. Acesso em: 15 de novembro de 2019. Citado na página 35.

27 BORONCZYK, T. *Working with Slim Middleware*. [S.l.], 2013. Disponível em: <https://www.sitepoint.com/working-with-slim-middleware/>. Acesso em: 16 de novembro de 2019. Citado na página 37.

28 FOWLER, M. *Patterns of Enterprise Application Architecture*. 1. ed. [S.l.]: Addison-Wesley Professional, 2002. Citado na página 38.

29 TERPIL, J. *Redux. From twitter hype to production*. [S.l.], 2015. Disponível em: <https://slides.com/jenyaterpil/redux-from-twitter-hype-to-production>. Acesso em: 28 de setembro de 2019. Citado na página 41.

30 WECK, S. *Developing modern offline apps with ReactJS, Redux and Electron*. [S.l.], 2019. Disponível em: <https://blog.codecentric.de/en/2017/12/developing-modern-offline-apps-reactjs-redux-electron-part-3-reactjs-redux-basics/>. Acesso em: 20 de novembro de 2019. Citado na página 42.