UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE AUTOMAÇÃO E SISTEMAS
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Mateus Conceição

Infraestrutura e aplicativo móvel para visualização de dados internos de computadores de bordo na agricultura

Florianópolis
2021

Mateus Conceição

Infraestrutura e aplicativo móvel para visualização de dados internos de computadores de bordo na agricultura

Relatório final da disciplina DAS5511 (Projeto de Fim de Curso) como Trabalho de Conclusão do Curso de Graduação em Engenharia de Controle e Automação da Universidade Federal de Santa Catarina em Florianópolis.
Orientador: Prof. Rômulo Silva de Oliveira, Dr.
Supervisor: Henrique Montagna Hartwig, Eng.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Conceição, Mateus
Infraestrutura e aplicativo móvel para visualização de dados internos de computadores de bordo na agricultura / Mateus Conceição ; orientador, Rômulo Silva de Oliveira, coorientador, Henrique Montagna Hartwig, 2021.
100 p.

Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Engenharia de Controle e Automação, Florianópolis, 2021.

Inclui referências.

1. Engenharia de Controle e Automação. 2. Aplicativo móvel. 3. Diagnóstico. 4. Agricultura. I. Oliveira, Rômulo Silva de. II. Hartwig, Henrique Montagna. III. Universidade Federal de Santa Catarina. Graduação em Engenharia de Controle e Automação. IV. Título.

Mateus Conceição

Infraestrutura e aplicativo móvel para visualização de dados internos de computadores de bordo na agricultura

Esta monografia foi julgada no contexto da disciplina DAS5511 (Projeto de Fim de Curso) e aprovada em sua forma final pelo Curso de Graduação em Engenharia de Controle e Automação

Florianópolis, 10 de maio de 2021.

Prof. Hector Bessa Silveira, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Rômulo Silva de Oliveira, Dr.
Orientador
UFSC/CTC/DAS

Henrique Montagna Hartwig, Eng.
Supervisor
Divisão de agricultura da Hexagon

Prof. Rodrigo Saad, Dr.
Avaliador
UFSC/CTC/DAS

Prof. Fabio Luis Baldissera, Dr.
Presidente da Banca
UFSC/CTC/DAS

AGRADECIMENTOS

À minha família, por me apoiar durante toda a graduação e garantir os meios para desenvolvimento de tal trabalho.

Ao supervisor deste trabalho, Henrique Hartwig, por estar sempre disponível para auxiliar no projeto e por me estimular constantemente no andamento do semestre.

Ao orientador, professor Rômulo de Oliveira, pela atenção desde o início do projeto e pelo rápido atendimento às demandas sempre que necessário.

Aos colaboradores e gerência da Hexagon por permitirem que eu desenvolvesse este trabalho e pelo direcionamento da implementação para obtenção de um produto de qualidade.

À Amanda Regina Rosa, por estar sempre ao meu lado e me apoiar durante essa caminhada.

Aos meus colegas do curso de Engenharia de Controle e Automação, que contribuíram para meu desenvolvimento pessoal e profissional ao longo dos últimos anos.

E, por fim, à Universidade Federal de Santa Catarina, pública e de qualidade, que me proporcionou experiências enriquecedoras e agregou conhecimentos essenciais para meu crescimento.

RESUMO

Com a expansão da digitalização no campo, soluções de *software*, como as desenvolvidas pela divisão de agricultura da Hexagon, se tornam cada vez mais complexas. Dessa maneira, aumentam-se as variedades de problemas durante as operações rurais. Além disso, com uma estrutura de suporte extensa, composta de diversos níveis de atuação, o diagnóstico e a solução dos problemas são, muitas vezes, postergados. Diante deste cenário, o objetivo geral deste trabalho é diminuir o tempo despendido para a solução de problemas encontrados em campo pelos usuários das soluções da Hexagon. Para isso, criou-se uma ferramenta em aplicativo móvel utilizando o *framework* de desenvolvimento de aplicativos híbridos *React Native*. Este se comunicará com uma infraestrutura de computação em nuvem *Amazon Web Services* a fim de interagir com o sistema a bordo do trator e coletar os seus dados de saúde, os quais serão apresentados ao usuário. Portanto, acredita-se que, descentralizando os dados para todos os envolvidos na cadeia de suporte, permite-se que a análise dos problemas ocorra em qualquer nível dela. Por meio do uso da metodologia ágil e da interação com os líderes das equipes de pesquisa e desenvolvimento, desenvolveu-se a ferramenta e toda a sua infraestrutura para resgatar remotamente e apresentar os dados internos do computador de bordo. Os resultados alcançados durante os testes preliminares foram consideravelmente positivos, cumprindo seu principal propósito. Embora nem todos os dados importantes para um diagnóstico completo tenham sido adquiridos pela ferramenta, acredita-se que, com a sua expansão, mais casos de problemas possam ser identificados.

Palavras-chave: Aplicativo móvel. Diagnóstico. Agricultura.

ABSTRACT

With the expansion of digitization in the agricultural field, software solutions, such as those developed by Hexagon's agriculture division, are becoming increasingly complex. Thus, it increases the variety of problems that happen during the rural operations. Besides, with a extensive support structure, composed by several levels, the diagnosis and the solution are, oftentimes, postponed. Due to this scenario, the main goal of this work is to reduce the time spent to find the solution for the problems found in field by the users of Hexagon's solutions. For this, it was created a tool in a mobile application using the framework for hybrid apps React Native. It will communicate with an Amazon Web Services cloud computing infrastructure in order to interact with the system on board the tractor and collect its health data, which will be presented to the user. Therefore, it is believed that, decentralizing the data to everyone involved in the support chain, the analysis is allowed to occur at any level of it. Through the use of the agile methodology and the interaction with the leaders of research and development teams, it was developed a tool and all its infrastructure to retrieve remotelly and present the internal data from the on-board computer. The results achieved during the preliminary test were considerably positive, fulfilling its main purpose. Although not all important data for a complete diagnosis have been acquired by the tool, it is believed that, with its expansion, more cases of problems can be identified.

Keywords: Mobile application. Diagnosis. Agriculture.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Ti ou TiX	Titanium
P&D	Pesquisa e Desenvolvimento
PDF	*Portable Document Format*
CAN	*Controller Area Network*
GNSS	*Global Navigation Satellite System*
ECU	*Electronic Control Unit*
D-Bus	*Desktop Bus*
GIS	*Geographic Information System*
iOS	Sistema operacional de produtos *Apple Inc.*
PWA	*Progressive Web App*
HTML	*HyperText Markup Language*
CSS	*Cascading Style Sheets*
JSX	*JavaScript XML*
API	*Application Programming Interface*
AWS	*Amazon Web Services*
S3	*Simple Storage Service*
IoT	*Internet of Things*
RDS	*Relational Database Service*
MQTT	*Message Queuing Telemetry Transport*
HTTP	*HyperText Transfer Protocol*
LoRaWAN	*Low Power Wide Area Network*
BD	Banco de Dados
REST	*Representational State Transfer*
SMS	*Short Message Service*
SNS	*Simple Notification Service*

XML	*eXtensible Markup Language*
ID	Identificador
UTI	Atualização de *software* do *Titanium*
CTI	Atualização de configuração do *Titanium*
SW	*Software*
JSON	*JavaScript Object Notation*
URL	*Uniform Resource Locator*

SUMÁRIO

1 INTRODUÇÃO

Neste capítulo será apresentado o grupo Hexagon e a sua divisão de agricultura, assim como o seu modelo de negócio no contexto da agricultura e as funções e responsabilidades da equipe de suporte dentre os processos da empresa. Além disso, será introduzido o problema a ser atacado pelo projeto em questão, juntamente com as motivações que levaram a sua escolha. Então, expõe-se a solução proposta para resolver tal problema, quais os objetivos do trabalho e a metodologia aplicada.

1.1 O GRUPO HEXAGON

A Hexagon é a líder global em sensores, *softwares* e soluções autônomas, fundada em 1992 e sediada em Estocolmo, Suécia, estando presente em mais de 50 países. O grupo Hexagon é separado em unidades de negócio, chamadas de “divisões”, em que cada uma delas é direcionado o foco de atuação à alguma área da tecnologia ou de aplicação, como, por exemplo, tecnologia geoespacial, segurança, mineração etc (HEXAGON, 2021).

O grupo como um todo possui o objetivo de utilizar da melhor forma os dados para alcançar um futuro que é além de automático, mas autônomo. Suas aplicações são extremamente direcionadas à aquisição, uso, gerenciamento e tratamento de dados com o intuito de trazer sucesso para seus clientes através da constante melhoria de performance das operações, aumentando a eficiência, qualidade e produtividade dessas.

Além disso, está em contato direto com os clientes e se compromete com a inovação contínua, visto que as demandas das empresas mudam a todo momento.

1.2 A DIVISÃO DE AGRICULTURA

A sede da divisão de agricultura fica localizada em Florianópolis, na qual foi desenvolvido o Projeto de Fim de Curso em questão. A empresa surgiu da união de três empresas que possuíam experiência no setor de agricultura de precisão: Arvus Tecnologia, ILab sistemas e parte da Hexagon *Geosystems* (AGRICULTURE, c2021). A divisão possui outras unidades localizadas em Ribeirão Preto (São Paulo) e Madrid na Espanha.

A atuação da empresa se dá em três grandes áreas: a área agrícola, principalmente da cana-de-açúcar; silvicultura; e através de parcerias com fabricantes de equipamentos agrícolas. Nas duas primeiras áreas, a divisão de agricultura da Hexagon atua em todas as fases do processo produtivo, ou seja, desde a preparação do solo até a colheita, além de permitir o monitoramento total da operação e auxiliar na logística. Já para os fabricantes, a empresa atua de forma a desenvolver e forne-

cer componentes de *hardware* (*drivers*, computador de bordo, piloto automático etc.) que são incorporados às máquinas da empresa parceira, a fim de melhorar o seu desempenho e agregar funcionalidades.

Dentre os maiores clientes de produção de cana-de-açúcar estão o Grupo São Martinho, a Raízen e a Adecoagro, da silvicultura, a Gerdau e a Suzano, e dos fabricantes, a *Shark*, a Jacto e a Jan.

Durante este documento, será referido à divisão de agricultura da Hexagon apenas como Hexagon, e, ao grupo como um todo, como Hexagon corporativa.

1.3 MODELO DE NEGÓCIO

A Hexagon implementa suas soluções de *software* relativas à operação em campo em um computador de bordo, chamado de *Titanium* (ou, apenas, *Ti* ou *TiX*). Esse computador de bordo se comunica com os componentes de *hardware* (*drivers* e atuadores), que se comunicam com o veículo, para que sejam atingidos os objetivos da aplicação, sejam eles de otimização, redução de custos, automatização de algum processo, entre outros. Cada conjunto de funcionalidades ou produto de *software* são vendidos separadamente de acordo com a demanda de aplicação do cliente por meio de ativações para liberação de acesso e utilização.

Geralmente, novas soluções são requisitadas por clientes que já possuem parceria para suprir suas demandas de determinado processo em campo. Essas soluções são encaixadas no plano de desenvolvimento das equipes de P&D de acordo com sua prioridade e são entregues em uma nova versão de *software*. Com o desenvolvimento completo e devidamente testado pela equipe interna de testes, faz-se a implantação em conjunto com diversos testes em campo para validar a solução no cenário e contexto do cliente. Além disso, realizam-se treinamentos com os operadores a fim de apresentar o funcionamento dessa nova solução.

1.4 EQUIPE DE SUPORTE E ENTREGA DE SOLUÇÃO

O projeto em questão foi desenvolvido na equipe de suporte e entrega de solução, que é encarregada de auxiliar no uso correto das soluções, triagem de *bugs*, implantação de algumas soluções e investigação de erros nas funcionalidades encontrados em campo.

Esses erros que ocorrem durante a operação no cliente são repassados e investigados pelos técnicos responsáveis por dado cliente e pela assistência técnica da Hexagon em um primeiro momento. Caso não consigam resolver o problema, um chamado é criado para a equipe de serviços de Ribeirão Preto, a qual analisa rapidamente o caso e aciona a equipe de suporte de Florianópolis se necessário. Também existe a possibilidade de criação de chamados diretamente para a equipe de Florianópolis.

Então, os chamados são alocados nas semanas seguintes de acordo com a estimativa de dificuldade e prioridade do problema.

Geralmente, o encarregado de analisar o caso faz a leitura da descrição do problema, conversa com o relator sobre o cenário em que o caso ocorreu, se necessário, e examina os arquivos de dentro do arquivo de diagnóstico e/ou confere as configurações da empresa e do computador de bordo. Identificada a causa do problema, faz-se a mudança necessária ou encaminha-se a solicitação de correção para a equipe responsável, a qual será validada posteriormente.

Esse processo de validação poderá ser melhorado através do desenvolvimento de uma ferramenta de apoio específica, pois, em conjunto com a atualização e o acesso remotos, pode-se enviar uma atualização com a correção e acompanhar os dados através da ferramenta para verificar o êxito da solução aplicada.

1.5 PROBLEMA

A princípio, apenas problemas maiores e que necessitam de um grau de conhecimento técnico mais elevado deveriam ser recorridos à equipe de suporte, por ser o último nível da cadeia de suporte. No entanto, muitos casos básicos e de fácil resolução levam algumas semanas para serem resolvidos devido a necessidade de seguir o processo completo de apoio ao cliente, o que pode causar descontentamento e receio por parte do cliente a respeito do(s) produto(s) vendido(s) pela Hexagon.

Além disso, atualmente a equipe de suporte tem bastante dificuldade em resolver alguns problemas que necessitam de constante interação com a aplicação do computador de bordo e, na maioria das vezes, leva semanas para ajudar o cliente a solucionar o mau funcionamento do produto. Também, alguns dados importantes são apenas disponíveis através de acesso remoto via terminal, o que torna inviável de serem analisados por algum técnico da empresa em campo, concentrando a análise do problema na equipe de suporte.

De maneira geral, grande parte dos problemas são levados durante toda a cadeia de suporte, aumentando consideravelmente o tempo para que o cliente tenha seu problema resolvido, intensificando a dependência de uma única equipe e centralizando a tomada de decisão em algumas pessoas que tem pouco ou nenhum conhecimento das especificidades da operação do cliente.

1.6 MOTIVAÇÃO

Como comentado anteriormente, há uma necessidade de melhoria do processo de suporte aos chamados dos clientes a fim de reduzir o tempo de espera por uma correção. Essa espera com o produto falho, dependendo do caso, é refletida no cliente na forma de máquinas e operadores parados, operação prejudicada ou comprometida,

perda de dados etc., o que traz grandes custos para a empresa e prejudica a reputação da Hexagon no ramo da agricultura.

Esses chamados de suporte são criados em um portal onde são adicionadas informações e uma descrição do problema, assim como fotos, vídeos e, na maioria dos casos, um arquivo de diagnóstico, o qual trás arquivos de execução e configuração do sistema (explicação mais detalhada na seção 2.1). Ao receber o chamado, a equipe analisa rapidamente o caso e aloca para investigar e resolver em alguma das semanas seguintes.

Dados os casos anteriores e o histórico de ocorrências da equipe, tentou-se reunir e elencar abaixo quais são os principais motivos que contribuem para o lentidão no processo e as maiores dificuldades atualmente ao se analisar um chamado de suporte.

- Em muitos casos, a descrição do caso é muito ampla, necessitando de algumas iterações com o relator para entender exatamente o cenário em que ocorreu o problema relatado.
- Ocorre ser preciso pedir o arquivo de diagnóstico, em casos onde ele é essencial para a análise e esse não foi disponibilizado no chamado. Até a equipe perceber a necessidade do arquivo, passaram-se, no máximo, uma semana e levará mais alguns dias até o recebermos de fato.
- Algumas das informações presentes no arquivo de diagnóstico não são disponíveis para visualização via interface do computador de bordo, apenas por comando no terminal, portanto não são acessíveis pelos técnicos.
- Muito é permitido ser visualizado por telas no computador de bordo, porém o operador precisa sair da tela de operação, o que faz com que a operação seja suspensa. Ou seja, atualmente, o operador é impossibilitado de monitorar alguns dados durante a operação, caso que pode ser importante em algumas situações.
- Alguns dados não são extraídos para o arquivo de diagnóstico, podendo ser visualizados apenas por linha de comando ou após instalação de configuração adicional.
- A coleta do arquivo de diagnóstico é feita em outra aplicação dentro do computador de bordo, fazendo com que algumas informações sejam perdidas ao finalizar a aplicação principal. Portanto, alguns problemas não são possíveis de investigar e identificar a causa.
- Necessidade de deslocamento físico de alguma pessoa para que seja feita a coleta do arquivo de diagnóstico para um *pendrive*.

- Devido aos processos da empresa, no melhor dos casos, o chamado é resolvido em uma semana ou, no caso de ser um *bug* no sistema, na próxima versão de *software*.

De modo geral, os principais problemas são perda de tempo com a coleta do arquivo de diagnóstico em casos simples que poderiam ser resolvidos em campo e a obrigatoriedade de seu uso para concluir algo sobre o caso.

1.7 SOLUÇÃO PROPOSTA

Como forma de reduzir o tempo gasto durante o suporte ao cliente, propôs-se o desenvolvimento de um aplicativo *mobile* que disponibiliza dados atualizados sobre a "saúde" do computador de bordo de interesse em algumas telas organizadas por tema. Ou seja, apresenta dados sobre a sua execução, armazenamento, configurações e também dados de algumas funcionalidades. Ademais, esses dados poderão ser exportados para um arquivo PDF para posterior análise e envio para outros interessados, se necessário.

Decidiu-se que a comunicação e transporte desses dados será feita através da nuvem de forma remota para que não seja necessário deslocar-se fisicamente até o computador de bordo a fim de coletar os dados, o que, atualmente, é necessário para coleta do arquivo de diagnóstico.

Será necessário desenvolver códigos que serão executados dentro dos computadores de bordo para resgatar os dados de interesse, pois, no momento, não existe nenhuma aplicação ou serviço implementado que possa suprir essa necessidade. No entanto, tem-se um serviço, chamado *Cloud Sync*, que realiza o envio de arquivos para a nuvem da *Amazon*, o qual será explicado com mais profundidade na seção 3.7.

Além disso, toda a arquitetura em nuvem deverá ser implementada para permitir a comunicação do *smartphone* com o computador de bordo.

Presume-se que, com mais informações, os técnicos compreenderão melhor o problema e irão reportar e descrever melhor o caso no chamado, reduzindo o tempo gasto para entendimento por parte da equipe de suporte. Além disso, pensa-se que o arquivo PDF exportado poderá ser útil ao se utilizar em conjunto com o arquivo de diagnóstico a fim de identificar a causa de determinado problema.

Com isso, acredita-se que melhoraria a responsividade do suporte para problemas pequenos em campo, reduziria o tempo de suporte como um todo, o que, por sua vez, reduziria os gastos do cliente e aumentaria sua satisfação com os produtos da Hexagon.

1.8 OBJETIVOS

De forma geral, o objetivo do projeto é reduzir o tempo de solução de chamados de suporte através da incorporação de uma ferramenta que busca agregar mais dados ao processo de forma mais ágil.

Com essa ferramenta, acredita-se que os técnicos poderão analisar o estado atual do computador de bordo e, com isso, resolver alguns problemas mais simples diretamente em campo, sem a necessidade de criar um chamado e seguir o processo de suporte completo.

O projeto como um todo tem como propósitos:

- Permitir o envio dos dados importantes para a nuvem e seu armazenamento durante determinado tempo a fim de poder ser requisitado posteriormente por algum usuário utilizando a ferramenta.
- Utilizar a comunicação através da nuvem, para que todos os colaboradores das equipes de suporte consigam ter acesso aos dados de determinado computador de bordo, desde que estejam autorizados.
- Integrar com uma base de dados já existente para os usuários se autenticarem e garantir a segurança dos dados dos computadores de bordo.
- Apresentar os dados do computador de bordo em uma interface de usuário amigável e organizada a fim de facilitar a compreensão e monitoramento do seu estado atual.
- Exportar os dados para um arquivo PDF para que possa ser guardado o estado atual do computador de bordo, o qual pode ser utilizado em um outro momento para evidenciar algum dado incoerente e comprovar a existência de algum problema, por exemplo.

1.9 METODOLOGIA

Para desenvolver esse projeto, em um primeiro momento, identificaram-se fraquezas da equipe de suporte em seus processos e fez-se uma reflexão sobre cada uma delas a fim de estimar o grau de dificuldade de algumas possíveis soluções.

Ao identificar uma fraqueza a ser atacada, organizou-se uma apresentação para os líderes da empresa contendo as motivações, objetivos e proposta de solução. Como finalidade dessa reunião, tinha-se a troca de ideias entre os participantes e o refinamento da solução, dando um direcionamento mais assertivo para o projeto. Além disso, buscou-se conversar com os prováveis usuários do sistema com o intuito de entender as suas demandas e verificar as suas opiniões acerca do projeto.

Com o projeto definido, prepararam-se protótipos, modelos e diagramas para que fossem definidas as tecnologias, os casos de uso e requisitos esperados para o projeto. Então, iniciou-se o estudo dessas tecnologias em conjunto com conversas com os desenvolvedores do setor de P&D da Hexagon, refinando cada vez mais o escopo e as formas de se implementar o sistema.

O desenvolvimento como um todo seguiu a metodologia de projetos ágil, a qual é baseada em ciclos de desenvolvimento. Em geral, tem-se reuniões periódicas a fim de retomar o que foi produzido no ciclo anterior e definir os objetivos do ciclo seguinte. Assim, tem-se um acompanhamento frequente sobre o andamento do projeto, o que permite, se necessário, uma mudança de direção mais facilmente, evitando possíveis retrabalhos no futuro.

1.10 ESTRUTURA DO DOCUMENTO

No capítulo 2, faz-se a apresentação do computador de bordo *Titanium*, dando mais detalhes quanto ao seu funcionamento, ao conteúdo do arquivo de diagnóstico e à forma como são feitos os acessos remotos no suporte ao cliente.

No capítulo 3, comenta-se sobre os aspectos teóricos que são importantes para o entendimento do projeto e que foram estudados durante a concepção do mesmo. Dentre os conceitos apresentados estão os tipos de programação *mobile*, detalhes sobre programação funcional e o *React*, os serviços da nuvem da *Amazon* utilizados no projeto e algumas tecnologias utilizadas no *software* do computador de bordo.

Já no capítulo 4, expõe-se o que foi definido quanto ao projeto arquitetônico do sistema como um todo, pontuando quais são seus requisitos e casos de uso, e definindo quais os dados a serem resgatados e apresentados ao usuário. Além disso, faz-se uma análise de qual plataforma será utilizada em contato com o usuário e uma pesquisa de mercado de aplicativos *mobile* no contexto da agricultura.

Nos capítulos 5, 6 e 7, trata-se, em detalhe, de cada um dos subsistemas que compõem o projeto, expondo os tópicos mais importantes de cada um deles.

No capítulo 8, avalia-se o projeto no âmbito do cumprimento dos requisitos levantados na modelagem, apresentam-se os testes realizados internamente e com os usuários finais, além de avaliar a solução como um todo após os testes.

Por fim, no capítulo 9, conclui-se os resultados que o projeto teve perante o cenário atual da empresa e são propostos trabalhos futuros para o projeto em questão, a fim de agregar valor à solução e amenizar seus pontos fracos.

2 O *TITANIUM*

Como comentado na seção 1.3, o *Titanium* é um computador de bordo onde são implementadas as soluções de *software* da Hexagon, nas quais o *Ti* tem a responsabilidade de se comunicar com a nuvem e com os outros equipamentos do veículo através da rede CAN (*Controller Area Network*); de tratar do posicionamento geográfico (GNSS); de armazenar dados e configurações; e de realizar o controle de mais alto nível. Para algumas das funcionalidades, atua em conjunto com componentes elétricos e eletrônicos, como *drivers* e ECUs (*Engine Control Unit*), os quais são encarregados de realizar o controle de baixo nível dos implementos e do piloto automático, além de adquirir dados de sensores. Na Figura 1, consegue-se visualizar como o *Titanium* fica disposto na cabine de um trator na agricultura, além do volante para controle de direção através do piloto elétrico.

Figura 1 – *Titanium* dentro da cabine de um trator.

Fonte – Documento interno da Hexagon.

O *Ti* possui uma tela de toque que permite que o operador do veículo agrícola navegue pelas diversas telas disponíveis, a fim de configurar os parâmetros da aplicação e das soluções, adicionar registros de veículos e implementos, ativar e desativar o piloto automático etc. Um exemplo de tela é apresentado na Figura 2 em que é mostrado a tela de operação, onde estão presentes a linha guia, suas paralelas e a área de aplicação ao centro, informações sobre o implemento na barra inferior, informações de conectividade no canto superior direito etc.

Internamente, o *Titanium* possui como sistema operacional uma distribuição

Linux onde diversos processos e serviços estão executando para atingir o objetivo da aplicação como um todo. Portanto, pode-se dizer que o computador de bordo possui grande capacidade de processamento e memória, sendo capaz de comportar a infraestrutura proposta para esse projeto.

A seguir, serão explicados mais a fundo o arquivo de diagnóstico e como são feitos os acessos remotos atualmente. Além disso, nas seções 3.6 e 3.7 são explicados o funcionamento da *D-Bus* e do *Cloud Sync*, respectivamente, os quais são conceitos internos ao computador de bordo importantes para a concepção do projeto.

Figura 2 – Exemplo de tela de operação do *Titanium*.

Fonte – Documento interno da Hexagon.

2.1 ARQUIVO DE DIAGNÓSTICO

Um arquivo de extrema importância para a análise de chamados é o arquivo de diagnóstico, pois este tem como finalidade retratar o estado atual do computador de bordo. Nele constam todos os arquivos de configuração atuais, todos os arquivos de pontos (arquivos .Ti)[1], os principais arquivos de *log* de execução e arquivos de *core*

[1] O arquivo .Ti é um *shapefile*, ou seja, arquivos utilizados por softwares de GIS (*Geographic Information System*) e armazenam informações geoespaciais vetorizadas, como pontos, polilinhas e polígonos. (ESRI, 1998) No caso do arquivo .Ti, armazena-se, a cada segundo, pontos na localização atual do veículo e, cada um dos pontos, possui atributos de operação, como velocidade, área aplicada etc.

dump[2].

Como comentado na seção 1.6, atualmente, esse arquivo só é possível de ser coletado presencialmente para um *pendrive* através de outra aplicação dentro do computador de bordo. Essa aplicação está disponível apenas em modo avançado, ou seja, é acessada apenas por algumas pessoas. Pelo fato de finalizar a aplicação principal, alguns *logs* são perdidos e, por ser exportado somente para um *pendrive*, faz com que o processo seja mais lento, pois é preciso se deslocar até o computador de bordo para coleta do arquivo.

Ademais, a análise do arquivo de diagnóstico é feita exclusivamente pelo time de suporte de Florianópolis, o que leva a uma dependência desnecessária e desperdício de tempo em casos onde o problema é mais simples.

2.2 ACESSO REMOTO

Atualmente, existe um serviço *Web* em que é possível visualizar a tela do computador de bordo de interesse e, se autorizado pelo operador, controlar e navegar através de toques na tela virtual. Esse serviço possui grande valor para os técnicos, visto que é possível realizar o auxílio aos clientes sem necessitar de deslocamento físico até onde o bordo se encontra, reduzindo significativamente o tempo de suporte.

No entanto, esse serviço de acesso remoto está muito instável e, praticamente, inutilizável nos últimos tempos devido à sua indisponibilidade, ao seu funcionamento imprevisível e à sua baixa confiabilidade.

Esses problemas existem em virtude de o sistema ter sido desenvolvido há muito tempo atrás e não ser mais mantido pelos desenvolvedores da equipe de *Cloud*. Os motivos pelos quais o projeto foi abandonado é por ser altamente custoso para a equipe entender como o sistema funciona, a fim de realizar as correções necessárias, além de não se ter certeza de que um dia o serviço chegasse a um estado aceitável de performance e disponibilidade.

Como forma de atender essa demanda dos técnicos, os colaboradores do P&D estão trabalhando em um novo sistema *Web* que irá substituir o serviço atual, porém contando com uma infraestrutura mais robusta e aprimorada. Com isso, o sistema será mais estável e atenderá melhor os interessados.

Acredita-se que, nesse novo serviço, será interessante acrescentar as informações de saúde do computador de bordo, para que o usuário possa acompanhar junto com a tela o estado atual do computador de bordo. Além do contexto de utilização

[2] *Core dumps* são arquivos que armazenam o estado da memória de determinado processo que foi interrompido de forma inesperada. Esses arquivos podem ser analisados através de programas depuradores (*debuggers*) em conjunto com a imagem de *debug* de tal processo (ARCHLINUX, 2021) . Na Hexagon, a equipe de suporte de Florianópolis executa a pré análise dos *core dumps* que ocorreram nas aplicações desenvolvidas pela empresa.

ser similar, o novo sistema poderá utilizar da infraestrutura desenvolvida neste projeto, apenas servindo como um outro tipo de interface para apresentação dos dados.

Outro tipo de acesso remoto utilizado durante o suporte a determinados problemas é o acesso direto ao terminal de comandos do computador de bordo. Dessa maneira, é possível executar qualquer tipo de comando, normalmente com as finalidades de identificar o problema ou realizar a correção. Porém, apenas colaboradores do P&D da unidade de Florianópolis podem realizar esse tipo de procedimento, por motivos de segurança. Além disso, alguns dados básicos só podem ser coletados através de comandos no terminal, não sendo acessível, portanto, para outros colaboradores.

3 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, faz-se uma contextualização acerca dos conceitos e tecnologias utilizados durante o projeto, os quais são necessários para o entendimento completo do trabalho desenvolvido. Dentre os tópicos abordados, comenta-se sobre: os tipos de programação para aplicativos móveis, o *framework React Native* em mais detalhes, alguns conceitos de programação funcional, a arquitetura de *software Flux*, a plataforma *Amazon Web Services* e os diversos serviços utilizados, o barramento *D-Bus* e um dos serviços do *software* do computador de bordo mais especificamente.

3.1 TIPOS DE PROGRAMAÇÃO *MOBILE*

Os aplicativos mobile são divididos em três grandes categorias de acordo com a forma que são executados dentro do dispositivo, as quais serão melhor detalhadas nas subseções seguintes.

3.1.1 Aplicativos nativos

Aplicativos nativos são aplicativos desenvolvidos utilizando a própria linguagem suportada pelo sistema operacional do dispositivo, ou seja, *Java* ou *Kotlin* para sistemas *Android* e *Objective-C* ou *Swift* para sistemas *iOS*.

Por serem executados nativamente no dispositivo, ou seja, não possuírem nenhuma camada de abstração, contam com o melhor nível de desempenho dentre as outras categorias, podem acessar todas as funcionalidades do dispositivo, além de ter a possibilidade de utilizar imediatamente as mudanças que ocorrem no sistema operacional. No entanto, é necessário codificar separadamente o aplicativo para cada uma das plataformas, sendo necessário mais tempo de desenvolvimento, maior dificuldade para corrigir problemas, atualizar o aplicativo e dar suporte aos usuários diversos.

3.1.2 Aplicativos *Web*

Aplicativos *Web* são aplicativos desenvolvidos de forma similar a uma aplicação *Web*, mas que possui visual próximo ao de um aplicativo. São executados em um navegador, portanto não necessitam de instalação, e as atualizações acontecem de acordo com a mudança da aplicação no servidor, facilitando a manutenção. Por serem aplicações Web, normalmente são mais fáceis de serem desenvolvidas pelo fato de ser um conhecimento mais difundido atualmente.

No entanto, para o aplicativo ter o formato e usabilidade característicos de um aplicativo nativo, são necessárias bibliotecas e pacotes adicionais que implementam certos componentes visuais ou, então, é preciso criar o componente do zero. Além disso, não se consegue disponibilizar funções *offline*, nem acessar funcionalidades na-

tivas do dispositivo, como geolocalização, *Wi-Fi*, armazenamento interno etc, limitando as capacidades do aplicativo.

Em 2007, a empresa *Google* introduziu o conceito de aplicativos chamados *Progressive Web Apps* (PWAs), os quais são aplicativos *Web* com conceitos e entidades adicionais (*App Shells* e *Service Workers*) a fim de se aproximar de um aplicativo nativo, como, por exemplo, a sua execução sem conectividade e envio de notificações. Porém, dependem do suporte dos navegadores e do sistema operacional às novas funcionalidades (LYNCH, 2016). Vale ressaltar que PWAs não são o mesmo que *Responsive Web Apps*, os quais são sites convencionais desenvolvidos que utilizam uma abordagem para apresentar a sua interface de acordo com o tamanho da tela do dispositivo, com o intuito de melhorar a usabilidade (TECHNOLOGIES, 2019).

3.1.3 Aplicativos híbridos

Os aplicativos híbridos são uma mistura das duas categorias anteriores, pois são escritos utilizando tecnologias Web (HTML, CSS e JavaScript) e não são executados através de um navegador, mas são apresentados a partir de uma entidade chamada *Webview*, a qual é encapsulada em uma aplicação base nativa (GRIFFITH, c2021). De maneira geral, a *Webview* é a parte do navegador que interpreta os arquivos *Web*, renderiza os seus componentes e apresenta os dados ao usuário. Ao estar inserido em uma aplicação nativa, pode-se acessar as funcionalidades nativas do dispositivo, abrindo mais possibilidades de implementação (CHINNATHAMBI, 2021).

Esse tipo de aplicativo possui a vantagem de possuir apenas um código e rodar em múltiplas plataformas, reduzindo o tempo e custos de desenvolvimento e manutenção. Também, como dito anteriormente, possuem a capacidade de acessar as funcionalidades do dispositivo e ser executado de modo *offline*.

Por possuir uma camada de abstração adicional, pode-se dizer que seu desempenho é prejudicado em relação aos aplicativos nativos. No entanto, essa diferença de desempenho vem diminuindo com o avanço da tecnologia e performance das camadas aplicadas ao sistema operacional. Ademais, em aplicações com até certo grau de complexidade essa diferença se torna próxima de ser imperceptível à nível de usuário comum.

Na maioria dos casos, são utilizados *frameworks* para facilitar o desenvolvimento desse tipo de aplicativo, portanto os desenvolvedores precisam esperar uma atualização do *framework* para poderem usufruir das novas atualizações do sistema operacional.

Considerando principalmente que o aplicativo será desenvolvido apenas por uma pessoa, que o tempo para construir o aplicativo é relativamente pequeno e que não é desejável restringir as possibilidades da implementação neste momento, decidiu-se que será utilizado algum dos *frameworks* para aplicativos híbridos no desenvolvi-

mento da parte *mobile* do projeto.

3.2 *REACT NATIVE*

React Native é um *framework open-source* para desenvolvimento de aplicativos multiplataforma criado em 2015 pelo *Facebook* e que utiliza a linguagem *Javascript* para codificar os aplicativos. Apesar de constar *Native* no nome, em muitos lugares o *React Native* é classificado como um *framework* para aplicativos híbridos por não ser codificado diretamente nas linguagens nativas (BROWN, 2021). Em contrapartida, em tempo de execução, o *React Native* cria os componentes de interface diferentemente para cada plataforma, assim a interface se comporta da mesma forma que componentes nativos, o que justifica o nome. Por exemplo, o componente "*Text*" é renderizado como um "*UITextView*" para o sistema *iOS* e como um "*TextView*" para o *Android* (SOURCE, c2021b).

Dentre as diversas opções de *frameworks* disponíveis para criação de aplicativos híbridos, escolheu-se o *React Native* pelo diferencial de sua interface se comportar e desempenhar como em aplicativos nativos. Além disso, o *framework* pode ser utilizado facilmente com códigos nativos em proporções diversas, não excluindo a possibilidade de desenvolvimento de partes da aplicação nativamente por necessitarem de melhor desempenho, por exemplo (SOURCE, c2021j). Ademais, utiliza apenas *Javascript* em todo seu código, o que facilita o aprendizado, é uma das linguagens de programação mais populares atualmente, com uma comunidade bastante presente e ativa recentemente, além de possuir uma documentação bem completa e organizada (CARBONNELLE, c2020).

3.2.1 JSX

Durante este documento serão apresentados códigos em *Javascript* da parte *mobile* utilizando JSX, a qual é uma extensão da linguagem para codificar os componentes *React* de forma mais legível que utilizando *Javascript* puro (SOURCE, c2021g).

As expressões JSX, durante a compilação, são transformadas em chamadas que retornam objetos *Javascript*, portanto podem ser utilizadas em estruturas condicionais, laços de repetição, retornos de função etc. Essas expressões permitem que sejam adicionados quaisquer trechos de código *Javascript* em seu interior (delimitado por chaves) como, por exemplo, o nome de alguma variável para que seja substituído pelo seu valor em um parâmetro.

Um exemplo de utilização do JSX é mostrado na Figura 3, em que é realizado a chamada da função "*formatName*" passando a variável "*user*" como argumento da função, contidos dentro do valor de um "*h1*". É possível notar também a atribuição de um elemento JSX a uma variável "*element*" que poderá ser utilizada posteriormente

na função de renderização do elemento.

Figura 3 – Exemplo de código utilizando JSX.

```jsx
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

```

Documentação do *ReactJS* (SOURCE, c2021g)

3.2.2 Estrutura

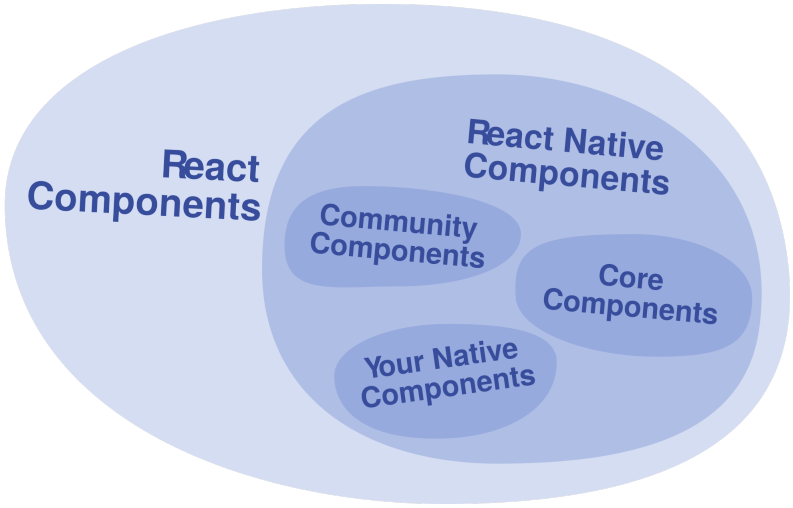
Assim como o *React*, a aplicação em *React Native* é composta por componentes, que podem ser organizados, aninhados e parametrizados de acordo com a aplicação, dividindo a interface em partes independentes e reutilizáveis (SOURCE, c2021a). Os componentes *React Native* podem fazer parte da biblioteca padrão (os chamados *Core Components*), de bibliotecas desenvolvidas pela comunidade com alguma finalidade específica, ou desenvolvida pelo programador, sendo que todos esses fazem parte dos componentes *React*. Essa hierarquia de componentes pode ser visualizada através da Figura 4.

Os componentes *React* podem ser codificados como componentes de função ou componentes de classe, os quais serão explicados com mais detalhes em seguida. Ambos recebem, em sua criação, propriedades (normalmente, refere-se apenas como "*props*") que definirão os parâmetros para criação do dado componente, as quais somente podem ser lidas.

3.2.2.1 Componentes de Classe

Componentes de classe têm como principal característica a presença de um estado e um ciclo de vida bem definido. O estado tem como função armazenar informações internas do componente durante seu ciclo de vida, como, por exemplo, o tempo atual de um temporizador.

Figura 4 – Hierarquia de componentes *React* e *React Native*.



Fonte – Documentação do *React Native* (SOURCE, c2021b)

O ciclo de vida de um componente é dado em três partes: criação ou montagem, atualização e desmontagem. Durante a fase de montagem, o seu método "*constructor*" é executado para inicializar o componente e seu estado e, em seguida, o método "*render*" que irá definir o que será mostrado na interface. Ao fim da montagem, o método "*componentDidMount*" é executado. Durante a execução do programa, podem acontecer atualizações no estado do componente, o que irá disparar os métodos "*render*" novamente e "*componentDidUpdate*" ao final da atualização. Durante a desmontagem do componente, apenas o método "*componentWillUnmount*" é executado. Existem alguns métodos que são executados além dos citados, mas que não são tão comumente usados (SOURCE, c2021c).

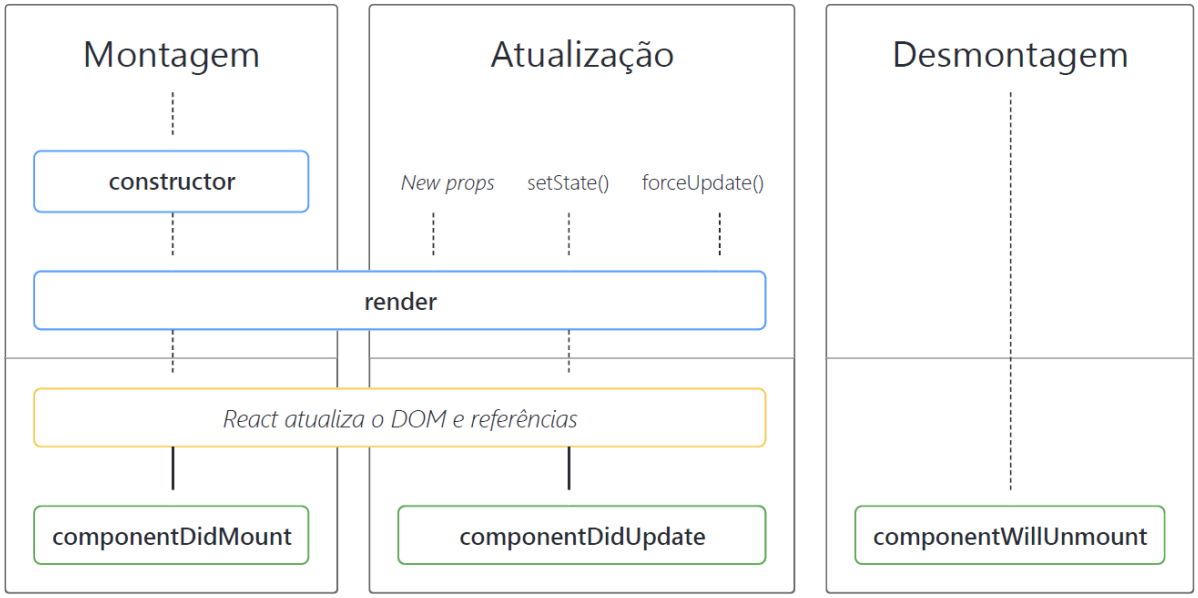
O diagrama mostrado na Figura 5 representa o ciclo de vida de um componente de classe no *React* conforme explicado anteriormente.

3.2.2.2 Componentes de Função

Componentes de função são funções *Javascript* que retornam um componente (normalmente uma expressão JSX) e, classicamente, não possuem estado nem um ciclo de vida. Porém, com a atualização 16.8 do *React* (0.59 do *React Native*), adicionou-se o conceito de *Hooks* para que seja possível utilizar estados e outros recursos de ciclo de vida sem a necessidade de criar um componente de classe. *Hooks* trazem diversas vantagens como uma melhor organização do código como um todo e a reutilização de lógicas envolvendo estados entre os componentes (SOURCE, c2021f).

Abaixo, nas Figuras 6 e 7, pode-se ver um exemplo de componente *React Native* implementado na forma de um componente de classe e de função, respectivamente,

Figura 5 – Ciclo de vida de um componente *React*.



Fonte – Documentação do *React* (SOURCE, c2021i)

para fins de comparação (SOURCE, c2021h).

3.3 PROGRAMAÇÃO FUNCIONAL

Programação funcional é um paradigma de programação declarativa no qual se projeta o *software* utilizando uma composição de funções puras (determinísticas), reduzindo o uso de estados compartilhados entre as partes da aplicação e efeitos colaterais (ELLIOTT, 2017).

Programação declarativa está relacionada com o que cada função faz ou retorna, ao contrário da programação imperativa, a qual está associada à forma de realizar determinada tarefa. Programando de forma declarativa faz com que o código fique mais limpo ao utilizar-se de expressões de mais alto nível, facilitando a sua compreensão. Muitas vezes, programação declarativa e funcional são tratadas como sinônimos (GOLLWITZER, 2020).

Um dos conceitos principais da programação funcional é o uso de funções puras, ou seja, que seguem os dois conceitos a seguir:

- Dadas as mesmas entradas (argumentos) da função, esta sempre retorna a mesma saída. Dessa forma, a chamada da função pode ser substituída pelo próprio valor do retorno da função, sem prejudicar o sentido do programa. Essa propriedade é chamada de transparência referencial (ELLIOTT, 2016).

Figura 6 – Exemplo de um componente de classe "*Cat*" em *React Native*.

```
class Cat extends Component {
    state = { isHungry: true };

    render() {
      return (
        <View>
          <Text>
            I am {this.props.name}, and I am
            {this.state.isHungry ? " hungry" : " full"}!
          </Text>
          <Button
            onPress={() => {
              this.setState({ isHungry: false });
            }}
            disabled={!this.state.isHungry}
            title={
              this.state.isHungry ? "Pour me some milk, please!" : "Thank you!"
            }
          />
        </View>
      );
    }
}

```

Fonte – Documentação do *React Native* (SOURCE, c2021h)

- Não causa efeitos colaterais durante sua execução, ou seja, não altera nenhum tipo de dado externo, inclusive os argumentos passados. Pode-se dizer que, por esse motivo, as funções puras preservam a imutabilidade do estado da aplicação, o que possibilita o armazenamento e acompanhamento do estado no tempo (ELLIOTT, 2016).

Com o uso de funções puras, muitos problemas de concorrência e dependência entre funções pelo uso de estados compartilhados são eliminados. Pode-se dizer que, através da transparência referencial e da imutabilidade do estado, as funções puras mantém o estado da aplicação determinístico, o que permite uma depuração de *bugs* e análise do fluxo do programa mais fácil. Além disso, funções que possuem essas propriedades se tornam independentes entre si, gerando um código mais maleável e adaptável à mudanças futuras.

Um outro conceito importante para a programação funcional é a composição de funções, ou seja, organizar e agregar funções básicas a fim de torná-las mais complexas para alcançar a funcionalidade desejada. Funções em *Javascript* são funções de primeira classe, ou seja, são tratadas como dados, o que permite ao programador atribuí-las a uma variável, usá-las como retorno de uma função etc. Portanto, é possível agregar funções e formar funções de ordem superior (*Higher order functions*), as

Figura 7 – Exemplo de um componente de função em *React Native*.

```
const Cat = (props) => {
    const [isHungry, setIsHungry] = useState(true);

    return (
      <View>
        <Text>
          I am {props.name}, and I am {isHungry ? "hungry" : "full"}!
        </Text>
        <Button
          onPress={() => {
            setIsHungry(false);
          }}
          disabled={!isHungry}
          title={isHungry ? "Pour me some milk, please!" : "Thank you!"}
        />
      </View>
    );
}

```

Fonte – Documentação do *React Native* (SOURCE, c2021h)

quais possuem como argumento uma função, retornam uma função, ou ambos.

O *React* é programado utilizando o paradigma de programação funcional devido aos seguintes fatores (CHIARELLI, 2018):

- Pode-se dizer que os componentes *React* são funções que tem como entrada as propriedades e como retorno parte da interface, sendo justamente essa a ideia na qual se baseiam os componentes de função.
- A aplicação é composta por componentes que são organizados entre si para criar novos componentes mais complexos. Em casos onde são feitas especializações entre componentes, pode-se pensar que ocorre herança entre eles, mas, na verdade, é considerado uma composição entre o componente generalizado com outros elementos de interface, gerando o elemento especializado.
- O conceito de imutabilidade é garantido pelo fato das *props* serem imutáveis, assim como as entradas das funções puras.
- As diretrizes de programação em *React* recomendam que sejam utilizados componentes sem estado (ou seja, componentes de função), tendo o comportamento de uma função pura. De maneira geral, tende-se armazenar os estados da aplicação em apenas um componente de nível hierárquico maior, evitando o uso de estados compartilhados.

- Possui o conceito de componentes de ordem superior, os quais são componentes que recebem como propriedade um componente e retorna um componente. Essa ideia é apenas uma adaptação do conceito de funções de ordem superior ou, para o caso de componentes de função, a mesma coisa.

Portanto, durante o projeto, serão seguidas as diretrizes e padrões do paradigma de programação funcional, visto que o *React* é baseado e implementado seguindo seus conceitos.

3.4 *FLUX* E *REDUX*

Como comentado anteriormente, distribuir o estado de uma aplicação com *React* em diversos componentes e compartilhar informações entre eles pode causar problemas de concorrência e inconsistência nos dados. Além disso, para aplicações grandes e com muitos componentes, a manutenção do código se torna bastante difícil por existirem diversas funções para realizar o sincronismo das informações entre os componentes, tornando a aplicação pouco escalável.

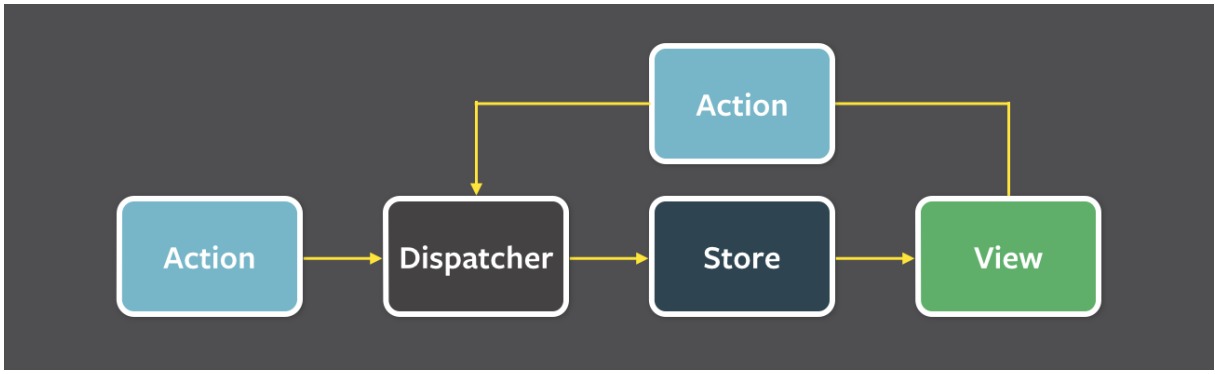
Como forma de resolver esse problema, uma recomendação é que os dados sejam armazenados em uma única fonte de informação, como um único componente no maior nível hierárquico da aplicação, o qual disponibiliza parte do estado para os outros componentes através das *props*.

Seguindo essa linha de raciocínio, o *Facebook* propôs a arquitetura de software *Flux* para ser utilizado em conjunto com o *React*. Nessa arquitetura, o fluxo de dados segue uma única direção, passando pelos seguintes elementos: *store*, *action*, *dispatcher* e *view* (ASIRI, 2018).

Os *stores* são os lugares onde são armazenados os estados da aplicação, isolados das *views*, as quais são os componentes que definem a estrutura da interface. *Actions* são as ações disparadas de acordo com algum evento proveniente das *views* (entrada do usuário, temporizador etc.). As ações são objetos *Javascript* que contém o tipo da ação (identificador) e os dados necessários para performar tal ação (*payload*) e são direcionadas ao *dispatcher*. O *dispatcher* possui as responsabilidades de designar para qual *store* a ação será direcionada e de isolar o *store* do restante da aplicação. Para fechar o ciclo, ao fim das mudanças no *store*, este envia um sinal para as *views*, que atualizam seu conteúdo de acordo com o novo estado atual e se re-renderizam. Exitem também os *Action Creators*, os quais são funções que recebem como parâmetros os dados de dada ação e retornam a ação já estruturada, com o intuito de facilitar a criação das ações. A Figura 8 ilustra o fluxo de dados no *Flux* (SOURCE, c2021d).

Já o *Redux* é uma biblioteca que implementa a arquitetura *Flux* para ser utilizada com diversos *frameworks* de desenvolvimento em *Javascript*, porém, normalmente, utiliza-se *Redux* em conjunto com *React*. Pelo fato de ser uma implementação do Flux,

Figura 8 – Fluxo de dados na arquitetura *Flux*.



Fonte – Documentação do *Flux* (SOURCE, c2021e)

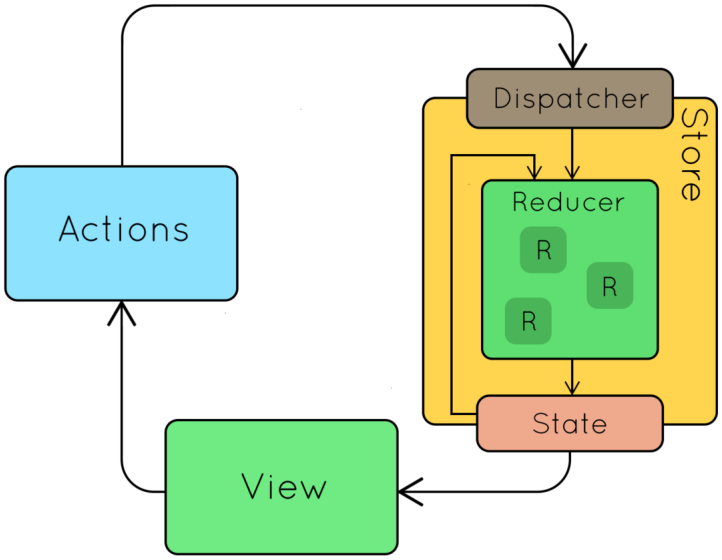
grande parte dos conceitos se mantém, porém possuem algumas divergências (BUNA, 2017):

- O *store* é único e imutável, ou seja, não é dividido em partes de acordo com o contexto como no Flux e as operações de mudança no estado são feitas por funções puras.
- O *store* não contém a lógica de negócio, ou seja, a "inteligência" de como se altera o estado conforme as ações não está contido no *store*.
- A entidade *dispatcher* foi removida e incorporada ao *store* na forma de um método da API disponibilizada pelo mesmo.
- As lógicas de negócio são implementadas pelos *reducers*, os quais se encontram dentro do *store*, podendo ser organizados e divididos de acordo com o contexto. Eles recebem o estado atual e a ação que foi disparada e retornam o novo estado através de funções puras.
- Adiciona-se o conceito de *selector*, que são funções para resgatar alguma informação do *store*, visto que a única forma de acessar o *store* é a partir de sua API.

Como pode ser visto na Figura 9, o fluxo de dados em *Redux* se diferencia, principalmente, pelo *store* englobar o *dispatcher* e conter diversos *reducers* em seu interior, como explicado anteriormente.

O *Redux* permite a adição de *middlewares* para customizar o comportamento do método *dispatch* a fim de adicionar funcionalidades entre o despacho da ação e sua chegada ao *reducer* correspondente. Os *middlewares* mais utilizados são para *logar* cada uma das ações executadas e o histórico do estado; emitir relatórios de *crash*; e

Figura 9 – Fluxo de dados na biblioteca *Redux*.



Fonte – Artigo (GUPTA, 2017)

utilizar de lógicas assíncronas a fim de resgatar dados de um servidor, por exemplo (REDUX, c2021).

É importante ressaltar que, pelo fato do *Redux* implementar as mudanças de estado através de funções puras, as aplicações que utilizam essa biblioteca são fáceis de depurar, pois cada mudança no estado é determinística e o histórico de mudanças pode ser acessado. Dessa forma, exitem *plug-in*s atualmente que permitem reproduzir o cenário problemático a partir do estado e das ações executadas, além de ser possível "viajar no tempo" da aplicação e analisar como o estado se comportou, permitindo a captura de *bugs* mais facilmente (SOURCE, c2021d).

3.5 *AMAZON WEB SERVICES* (AWS)

A *Amazon Web Services* (AWS) é uma plataforma que disponibiliza centenas de serviços computacionais em nuvem, dos quais alguns são utilizados pela Hexagon para permitir que suas soluções sejam implementadas (SERVICES, c2021t).

Assim como outras plataformas de computação em nuvem, a AWS disponibiliza seus serviços de forma transparente para o usuário, de forma que ele não precisa se preocupar com questões de implementação, escalabilidade, segurança, disponibilidade, entre outros.

Além disso, o custo de uso de seus serviços é proporcional à demanda e de acordo com a categoria de tecnologia que o serviço pertence. Por exemplo, serviços de armazenamento irão custar mais à medida que a quantidade de dados armaze-

nados aumenta, e serviços de computação serão mais caros de acordo com o custo computacional utilizado (tempo e memória).

Dentre os serviços disponíveis e usados pela Hexagon, os que serão utilizados no contexto do projeto são o *Simple Storage Service* (S3), *Internet of Things* (IoT Core), *Lambda*, *Relational Database Service* (RDS), *API Gateway*, *Cognito* e *DynamoDB*, cujas funções serão explicadas nas seções a seguir.

3.5.1 *Simple Storage Service* (S3)

O *Simple Storage Service* (S3) consiste em um serviço de armazenamento de dados simples em nuvem líder em seu setor pelo fato de oferecer fortes vantagens com relação à confiabilidade, escalabilidade, desempenho e facilidade de uso e acesso (SERVICES, c2021s). Por esse motivo, é utilizado por empresas de tamanhos diversos em suas aplicações de contexto e complexidade variados. De acordo com a quantidade de acessos e a finalidade do armazenamento, o S3 possui categorias especiais a fim de atender melhor a necessidade da aplicação (SERVICES, c2021o).

O armazenamento no S3 é dividido em *buckets*, os quais podem ser divididos em pastas e subpastas. No contexto da Hexagon, existe um *bucket* para armazenar os arquivos enviados pelo *Cloud Sync* (o qual será detalhado na seção 3.7) separados de acordo com o número de série do computador de bordo. Portanto, aproveitou-se esse mesmo *bucket* para guardar os arquivos enviados contendo os dados do bordo.

3.5.2 *IoT Core*

O *IoT Core* é um serviço de *Internet* das Coisas que disponibiliza toda a infraestrutura e segurança necessária para a conexão de um dispositivo à *Internet* por meio dos protocolos de comunicação mais utilizados para este fim (*MQTT*, *HTTP*, *WebSockets* e *LoRaWAN*). A infraestrutura oferecida tem a capacidade de comportar, simultaneamente, bilhões de dispositivos e trilhões de mensagens (SERVICES, c2021r).

A partir de um mecanismo de regras é possível rotear as mensagens recebidas pelos dispositivos para os outros serviços da AWS a fim de armazená-las e processá-las automaticamente.

O custo é definido de acordo com as funcionalidades utilizadas nesse serviço, sendo, basicamente, calculado levando em consideração o número de mensagens enviadas, o número de regras disparadas e ações realizadas em cada uma dessas regras (SERVICES, c2021j).

Para a Hexagon, o *IoT Core* é um serviço essencial para implementar a maioria das soluções baseadas em telemetria e monitoramento, pois permite o envio constante de informações sobre a operação dos veículos em campo. Com essas informações

é possível realizar os tratamentos, análises e consultas desejadas e apresentar ao cliente para que este possa tomar decisões mais assertivas.

No contexto do projeto, empregou-se o *IoT Core* para permitir a comunicação do aplicativo *mobile* com os computadores de bordo em campo.

3.5.3 *Lambda*

O serviço *Lambda* é um serviço de computação sem servidor (*serverless*) para executar códigos sem se preocupar com a criação e gerenciamento de máquinas servidores e com a quantidade de tempo e memória necessário para sua execução. Além disso, garante alta disponibilidade e escalabilidade, com alocação dinâmica de recursos de acordo com a demanda do código (SERVICES, c2021d).

As funções podem ser executadas automaticamente através de eventos oriundos de outros 140 serviços da *Amazon* ou diretamente pela interface do site da AWS (SERVICES, c2021p). A cobrança é calculada de acordo com a quantidade de memória alocada, tempo de computação gasto e quantidade de execuções feitas (SERVICES, c2021k).

Para facilitar o desenvolvimento e manutenção das funções *Lambda*, a AWS disponibiliza os *logs* de execução das funções no serviço *CloudWatch*, além de mostrar quantas solicitações foram feitas, quantas estão suspensas, latência etc.

É possível utilizar as Camadas *Lambda* (ou *Lambda Layers*) para compartilhar trechos de código entre as funções *Lambda* a fim de reutilizar o código, facilitar a atualização das funções dependentes dessa camada, e reduzir o tamanho do pacote de implantação (KOKJE, 2019).

No âmbito do projeto, utilizou-se as funções *Lambda* para realizar os tratamentos necessários para cada mensagem de saúde enviada pelo computador de bordo e permitir o fluxo de dados entre os serviços utilizados.

3.5.4 *Relational Database Service* (RDS)

O *Relational Database Service* (RDS) é um serviço para configuração e operação de bancos de dados relacionais em nuvem, com suporte para os mecanismos de BDs relacionais *Amazon Aurora*, *MySQL*, *MariaDB*, *Oracle*, *SQL Server* e *PostgreSQL*. Assim como outros serviços da *Amazon*, o RDS conta com alta disponibilidade e alocação dinâmica de recursos para atender as demandas das aplicações (SERVICES, c2021e).

Além disso, o RDS é responsável pelo gerenciamento do banco de dados, podendo fazer *backups* automaticamente, mantém o *software* do BD sempre atualizado e facilita a utilização de replicação do banco de dados (SERVICES, c2021n).

O custo desse serviço é baseado, principalmente, na região em que será implantada a instância do banco de dados, na quantidade de horas que o banco está

rodando, na quantidade de instâncias utilizada e no armazenamento reservado para a instância (SERVICES, c2021i).

Para esse projeto, fez-se uso do RDS para indexação dos arquivos em uma tabela para manter informações sobre cada um dos pacotes de dados enviados pelo computador de bordo, as quais podem ser acessadas através de chamadas pelo aplicativo.

3.5.5 *API Gateway*

O serviço *API Gateway* permite que sejam criadas, mantidas e monitoradas interfaces *API RESTful* e *API WebSocket*, as quais permitem o acesso de aplicações externas a alguma funcionalidade em nuvem. É de responsabilidade do serviço cuidar do controle de acesso, autorização, garantir o atendimento de até centenas de milhares de chamadas simultâneas etc (SERVICES, c2021a).

O custo de uma *API RESTful* é proporcional ao número de chamadas, à quantidade de memória *cache*, ao tempo de armazenamento em cache e à quantidade de dados de saída. Já para uma *API WebSocket*, o custo é definido conforme a quantidade de mensagens enviadas e recebidas e o tempo de conexão. Cobranças adicionais são feitas ao utilizar outros serviços a partir das interfaces (SERVICES, c2021f).

Com o *API Gateway* é possível criar uma interface que irá executar funções *Lambda*, executar chamadas à *endpoints* HTTP de aplicações hospedadas ou não em outros serviços de computação em nuvem, entre outras funções (SERVICES, c2021l). Dessa forma, no projeto em questão, utilizou-se esse serviço para permitir chamadas às funções *Lambda* através do aplicativo *mobile* para acessar os dados armazenados em nuvem e enviar comandos.

3.5.6 *Cognito*

O serviço *Cognito* realiza o gerenciamento de usuários, *login* e controle de acesso de usuários, além de oferecer suporte a *login* com provedores de identidade social (como o *Google*, por exemplo) (SERVICES, c2021b). Possui capacidade para milhões de usuários e está compatível com diversas certificações de qualidade e segurança (SERVICES, c2021q).

Quanto às cobranças, o custo desse serviço depende da quantidade de usuários ativos mensais, ou seja, usuários que, dentro de um mês, foram cadastrados, efetuaram *login*, atualização de *token* ou alteraram a senha. Além disso, há custo adicional para utilização de Autenticação Multi Fator por SMS de acordo com as cobranças do serviço de notificações da AWS (*Amazon* SNS) (SERVICES, c2021g).

Na Hexagon, existe um grupo de usuários já cadastrados nesse serviço e o aplicativo *mobile* desenvolvido recorre a ele para garantir a autenticação dos usuários.

3.5.7 *DynamoDB*

O *DynamoDB* é um serviço de bancos de dados *NoSQL* no estilo chave-valor e documentos, ou seja, não segue os padrões de bancos de dados relacionais e armazena os dados de alguma entidade a partir de um valor dado como chave. Pelo fato de ser organizado dessa forma, possui a vantagem de ser extremamente rápido em suas consultas dadas em qualquer escala (SERVICES, c2021c).

Similar ao serviço de funções *Lambda*, o *DynamoDB* tem o benefício de ser um banco de dados sem servidor e escalável automaticamente de acordo com a demanda, fato que facilita a implantação de soluções e manutenção dos dados mais facilmente se comparado com os bancos providos. Além disso, a AWS cuida do gerenciamento dos dados, replicação, garantia das transações etc (SERVICES, c2021m).

Os gastos desse serviço é dado de acordo com o plano que o usuário escolher com base na aplicação que irá utilizar o serviço: sob demanda ou provisionado. Para o plano sob demanda, não são definidas as quantidades de transações por segundo, quantidade de dados a ser armazenado etc, as quais são determinadas *a priori* no plano provisionado. Os gastos principais levam em consideração as unidades de solicitação ou capacidade de escrita e leitura, os quais são baseadas na consistência dos dados armazenados, sendo que maior consistência gera mais custos. Além disso, pode-se utilizar recursos adicionais que agregam ao valor final, como tabelas globais, *backups*, restaurações etc (SERVICES, c2021h).

Na Hexagon, existem tabelas implementadas no *DynamoDB* a fim de relacionar os usuários do sistema de acesso remoto comentado na seção 2.2 com os números de série que esse usuário pode ter acesso. Essa relação é feita através de grupos de usuários, sendo que cada usuário pode pertencer a diversos grupos. Esses dados serão utilizados a fim de controlar o acesso dos usuários quanto à inspeção dos computadores de bordo.

3.6 *D-BUS*

Como comentado no capítulo 2, o *Titanium* é dotado de um sistema operacional, no qual a transmissão de dados entre os processos é feita por meio da *D-Bus*.

A *D-Bus* é um sistema de comunicação entre as aplicações de um sistema, sendo esta feita através de troca de mensagens em um barramento lógico (FREEDESKTOP.ORG, 2020). Por se tratar de um barramento, tem-se a vantagem de se padronizar a forma de comunicação entre as aplicações, diminuindo possíveis erros de interpretação, além de reduzir o número de pontes necessárias entre cada uma dos processos em sistemas de maior porte.

A *D-Bus* é constituída de diversos barramentos que representam as aplicações do sistema, os quais são identificados pelo seu nome. Esse nome, em um primeiro

momento, é definido aleatoriamente pela própria *D-Bus*, mas que pode ser trocado para outro nome mais representativo e legível por meio de uma requisição à *D-Bus* (COCAGNE, 2013).

Cada barramento possui um ou mais objetos, que representam os objetos da aplicação em questão, sendo identificados pelo seu caminho dentro do barramento. O caminho do objeto possui estrutura similar ao de um sistema de arquivos, permitindo organizar os objetos de forma hierárquica se necessário.

Cada objeto pode possuir diversas interfaces, as quais definem quais são os métodos e sinais que estão disponíveis para as outras aplicações. Os métodos e sinais são descritos através de assinaturas, de forma semelhante a algumas linguagens de programação, descrevendo a quantidade e tipo dos parâmetros e do retorno. Essas assinaturas são definidas em um arquivo XML que pode ser consultado em tempo de execução através da interface padrão a cada objeto, chamada "*org.freedesktop.DBus.Introspectable*".

Os métodos seguem o mecanismo de chamada remota de procedimentos, ou seja, são feitas de forma síncrona, em que a aplicação cliente espera pelo retorno do método. Já para os sinais, utiliza-se o mecanismo de *publish-subscribe*, ou seja, as aplicações cliente interessadas em determinado sinal, devem se inscrever no mesmo. Quando o sinal disparar, todos os inscritos receberão a mensagem.

No caso desse projeto, utilizou-se a *D-Bus* para inserir o processo que irá resgatar os dados de interesse e preparar o arquivo compilado, mas também para invocar métodos das interfaces *D-Bus* a fim de resgatar diversas informações dos processos em execução no computador de bordo.

3.7 *CLOUD SYNC*

Como as soluções da Hexagon possuem foco na obtenção de dados das operações em campo, faz-se o uso de estruturas computacionais em nuvem para processamento e armazenamento desses dados. Esses dados são enviados através de um serviço chamado *Cloud Sync*, que está constantemente executando no sistema do computador de bordo. Como o nome sugere, esse serviço busca manter a sincronia dos dados que estão no bordo com os dados armazenados em nuvem.

Além disso, possui um barramento na *D-Bus*, onde disponibiliza um método que faz o envio de um pacote compactado contendo os arquivos desejados. Esse método será fundamental para envio dos arquivos de inspeção para a nuvem e, consequentemente, para o *smartphone* do usuário do aplicativo.

Os pacotes enviados pelo *Cloud Sync* possuem destino em um *bucket* específico no S3, onde são organizados de acordo com o número de série do computador de bordo. Esses arquivos podem ser consumidos por diversas aplicações e serviços implementados na nuvem.

4 PROJETO ARQUITETÔNICO

Neste capítulo é apresentado o sistema chamado de *TiX Inspector* sob o ponto de vista mais geral, expondo quais são seus casos de uso, os requisitos funcionais e não funcionais do sistema e quais os dados importantes a serem apresentados ao usuário. Além disso, realiza-se uma discussão acerca da plataforma a ser utilizada para interfacear com o usuário, expõe-se uma breve pesquisa de mercado de aplicativos móveis no cenário das empresas de agricultura e uma visão geral da arquitetura do sistema.

Vale ressaltar que o autor deste trabalho contou com o auxílio dos colaboradores do setor de P&D da Hexagon para tomar decisões de projeto e assimilar o funcionamento tanto dos sistemas compreendidos dentro do computador de bordo, quanto dos serviços e estrutura da nuvem AWS. No entanto, o desenvolvimento em si do sistema foi completamente produzido pelo autor.

4.1 CASOS DE USO

O diagrama de casos de uso tem o objetivo de representar quais são os atores que interagem com o sistema e a relação que eles terão com as funcionalidades definidas dentro do escopo do sistema (SOFTWARE, c2021).

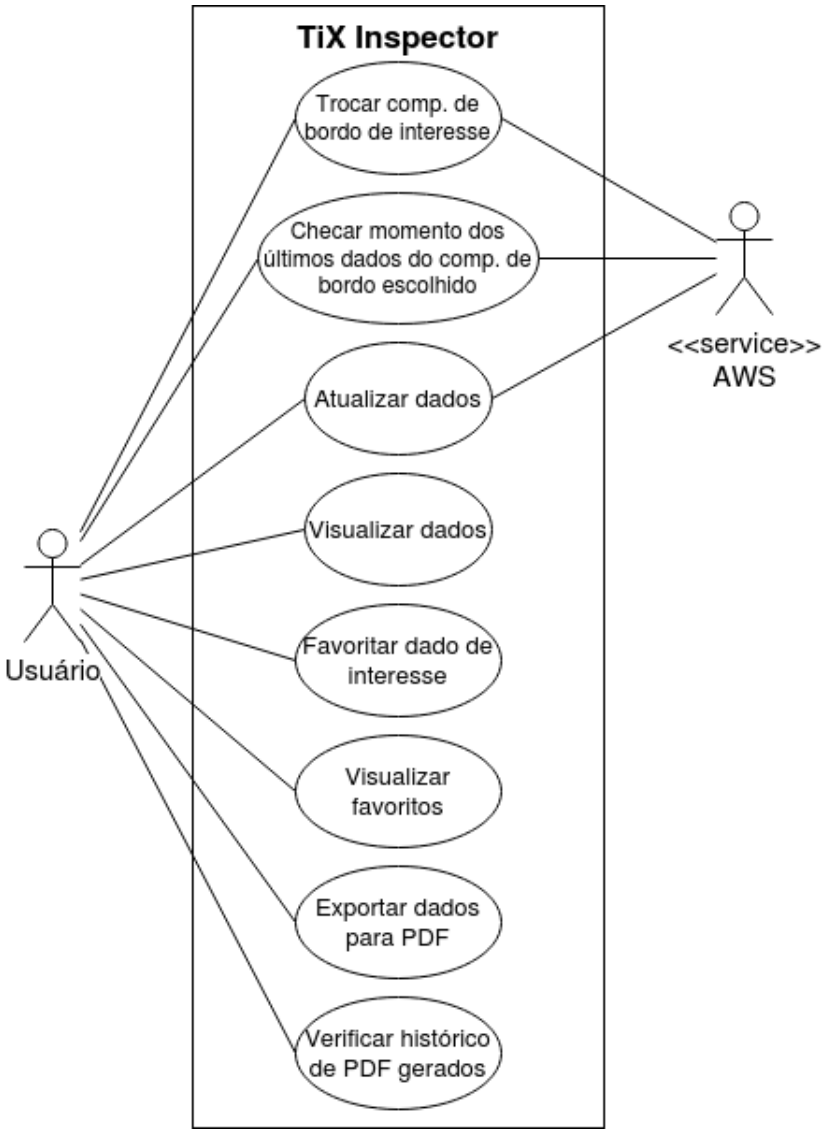
Tendo isso em mente, definiu-se que os atores humanos do sistema desempenham o mesmo papel de “usuário”, pois podem ter acesso as mesmas funcionalidades. Porém, o sistema se comunica com os serviços na nuvem, os quais são representados por um ator chamado “AWS”, simbolizando o sistema em nuvem como um todo.

No aplicativo proposto, o usuário é capaz de escolher o computador de bordo que deseja inspecionar em qualquer momento, e checar a última vez que o computador de bordo enviou dados sobre sua saúde. Com relação aos dados, o usuário pode visualizá-los na interface do aplicativo, favoritar algum dado de interesse para acesso mais rápido posteriormente, visualizar os dados favoritados separadamente e atualizar os valores sendo mostrados quando achar necessário. Além disso, o usuário tem a possibilidade de exportar os dados atuais para um arquivo PDF para uso posterior, similar a uma foto do estado atual do computador de bordo, os quais são armazenados localmente no *smartphone*, podendo ser consultados no futuro. Quanto ao ator “AWS”, esse está ligado com as funcionalidades de checar último envio e atualizar os dados atuais do computador de bordo, pois atuará como meio de comunicação até ele. Além disso, o ator “AWS” está relacionado com a ação de trocar computador de bordo de interesse, pois será consultado a fim de verificar quais os computadores de bordo são permitidos ao usuário.

Cada funcionalidade citada acima é apresentada na Figura 10 na forma de uma oval conectadas aos atores do sistema, sendo que o retângulo representa o escopo de

funcionalidades que o sistema irá comportar, e os bonecos palito retratam os atores do sistema.

Figura 10 – Casos de uso do sistema



Fonte: Arquivo pessoal.

4.2 REQUISITOS DO SISTEMA

Os requisitos do sistema podem ser divididos em dois grupos: os requisitos funcionais e os requisitos não funcionais. Os primeiros têm como finalidade definir o que o sistema deve ser capaz de executar a fim de cumprir a sua função. Normalmente, estão altamente ligados com as funcionalidades do sistema. Já os requisitos não funcionais estabelecem a forma como serão alcançados os requisitos funcionais,

impõem restrições ao sistema, define tecnologias a serem suportadas e/ou utilizadas etc (CANGUÇU, 2021).

Tendo em vista os casos de uso do sistema apresentados anteriormente, definiram-se os requisitos funcionais abaixo.

1. Utilizar de recursos gráficos para apresentar dados do pacote enviado pelo bordo.
2. Atualizar interface com dados mais recentes quando o usuário requisitar.
3. Separar internamente os dados de cada computador de bordo para que seja possível a troca do computador de bordo de interesse.
4. Permitir que o usuário favorite algum dado que tenha maior interesse, com o intuito de inspecioná-los mais facilmente.
5. Permitir que o sistema requisite um novo pacote de dados, baixe pacotes de dados novos e verifique a data e hora do último pacote do computador de bordo.
6. Integrar com sistema de login já existente nos sistemas da Hexagon a fim de utilizar o mesmo *pool* de usuários.
7. Exportar dados para PDF.
8. Manter dados e arquivos no armazenamento local para que o usuário retome a seção a partir do estado em que parou na última seção e possa resgatar os arquivos PDF exportados.
9. Dentro do computador de bordo, resgatar os dados propostos na seção 4.3.

Quanto aos requisitos não funcionais, tem-se o seguinte:

1. Ser compatível com sistemas operacionais *Android* e *iOS*, com a intenção de permitir que o maior número de usuários possa usufruir das funcionalidades do aplicativo durante o processo de suporte.
2. Comunicar com infraestrutura desenvolvida na AWS, tendo em vista que é a plataforma de computação em nuvem utilizada pela empresa.
3. Organizar os dados de acordo com seu contexto, para maior facilidade em encontrá-los ao navegar pelo aplicativo.
4. Possuir arquitetura de software modular e escalável, permitindo expansões futuras com menor esforço.
5. Dispor de uma interface de usuário similar a aplicativos atuais para maior aceitação por parte dos usuários e maior aderência da solução no processo da empresa.

4.3 INFORMAÇÕES A SEREM COLETADAS

Um tópico de grande importância é a definição de quais dados serão coletados dos sistemas e serviços executando no computador de bordo. Os dados devem ser estabelecidos tendo em vista o objetivo de auxiliar na investigação de problemas de execução e complementar o arquivo de diagnóstico.

No contexto mais geral, adicionou-se as seguintes informações:

- Informações básicas como, por exemplo, o **ID da placa base** (*baseboard ID*) e o **número de série** para conferências básicas em um primeiro momento. Não é um caso recorrente, mas pode ser que o *baseboard ID* esteja desatualizado nas bases de dados da empresa, gerando alguns erros de identificação do equipamento.

- O **horário atual** e o **fuso horário** configurado são úteis para garantir que não há problemas de defasagem no horário e para acompanhar o horário dos arquivos de *logs* de forma correta.

- Informações de **armazenamento interno** são importantes para acompanhamento do espaço disponível para guardar os arquivos que são gerados durante a execução do sistema, visto que já houveram problemas no passado relacionados à falta de espaço.

- Dados sobre os ***core dumps*** que ocorreram anteriormente e estão guardados no computador de bordo, como horário, data e tamanho do arquivo. Esses dados são importantes de serem coletados, pois não há uma forma prática de consegui-los a não ser por acesso remoto ao terminal ou coletando um arquivo de diagnóstico.

- Quais **ativações de software** estão ativas no bordo com o intuito de conferir se estão de acordo com a venda, pois já ocorreu de em alguns computadores de bordo ter inconsistência com o que havia sido contratado.

- Qual a **versão de *software* (*UTI*) e configuração (*CTI*)** está instalada no momento e qual a previamente instalada. Esses dados são comumente utilizados para identificar incompatibilidade de versões e *CTI*s mal construídos, por exemplo.

- Quais os **processos em execução** no momento e suas características, como, por exemplo, porcentagem de processador e memória ocupados. Esses dados podem ser utilizados para conferir se há algum processo com alto consumo dos recursos de *hardware*, acarretando travamentos e lentidão.

Das informações de conectividade, buscou-se os seguintes dados:

- Do âmbito mais geral, resgatou-se as **conexões salvas** e os **dispositivos disponíveis** e instalados, pois houve problemas com conexões acumuladas, acarretando em problemas de conexão.

- Dados do **Wi-Fi**, como dispositivos disponíveis (*Wi-Fi* interno e externo), qual a rede conectada atualmente e a força do sinal, a fim de identificar inconsistências com o que deveria ser configurado por *CTI* e monitorar o sinal da rede em que o bordo está.

- Quanto à conectividade móvel, buscou-se dados sobre o **modem instalado** (modelo, interface etc.), **configurações da rede** (usuário e senha da operadora, por exemplo) e a **força do sinal**. Todos esses dados são bastante importantes por motivos semelhantes ao *Wi-Fi*.

- Coletou-se dados de **configuração do GNSS**, **posição atual**, **qualidade do sinal**, entre outros dados importantes para certificar a garantia de qualidade de soluções que usam a posição geográfica. Como existem diversas configurações e sinais disponíveis para uso com suas especificidades, é comum ocorrer erros no processo de configuração do GNSS.

- Dados sobre a conectividade com a nuvem e sua sincronia são importantes, pois envolvem envio de dados de operação do cliente, os quais são considerados valiosos e essenciais pelas empresas. Consegue-se verificar se o computador de bordo está de fato **conectado** e os **arquivos que estão pendentes**, ou seja, estão armazenados para posterior envio. Além disso, é possível verificar a existência de **certificados** para autenticação com os sistemas da *Amazon*, o que não consta no arquivo de diagnóstico nem em tela alguma. Ademais, é possível identificar o **estado de todos os arquivos de operação (.Ti)** gerados pelo bordo.

Para exemplificar a coleta de algum dos produtos vendidos pela empresa, escolheu-se o monitoramento, pois está presente em quase todos os clientes atualmente. Dessa funcionalidade, colheu-se dados de **configuração**, como tipo de veículo, centro de custo, função etc. Além disso, resgataram-se os **valores dos sensores** que estão dispostos na tela de monitoramento, para possibilitar a sua verificação durante a operação. A tela de monitoramento é apresentada na Figura 11 a fim de elucidar o que foi dito.

4.4 DEFINIÇÃO DA PLATAFORMA

Com o intuito de definir qual a melhor plataforma para se implementar a interface com o usuário que irá mostrar os dados, comparou-se as opções de implementação

Figura 11 – Exemplo de tela de monitoramento

Fonte: Arquivo pessoal.

diretamente no computador de bordo, em uma plataforma *Web* ou em um aplicativo *mobile* de acordo com alguns critérios.

Quanto às funcionalidades que poderiam ser suportadas em cada uma das plataformas, viu-se que a única a permitir que todas elas sejam implementadas foi a plataforma *mobile*. Nela, é possível realizar a conexão remota através da conectividade do celular (seja *Wi-Fi* ou móvel) com os serviços em nuvem, e conectar-se à computadores de bordo diferentes rapidamente. Além disso, utilizando um aplicativo *mobile*, permite-se uma possível ampliação do escopo da aplicação para permitir conexão local (através de alguma tecnologia de pareamento, como rede local *Wi-Fi* ou *Bluetooth*), na qual não necessitaria de conectividade com a *internet* tanto pelo *smartphone* quanto pelo computador de bordo.

Pelo fato de utilizar a *internet* como meio de transmissão dos dados, as plataformas *Web* e *mobile* permitem que múltiplos usuários possam realizar a inspeção de um mesmo computador de bordo de forma simultânea. Essa funcionalidade pode ser interessante, por exemplo, para acompanhamento do estado de determinado computador de bordo tanto por alguém da equipe de suporte de Florianópolis, quanto por algum técnico em campo, atuando em conjunto para solucionar o problema.

Percebeu-se que, pelo fato de ser um dispositivo à parte e possuir uma tela separada, a plataforma *mobile* e *Web* permitem que os usuários possam inspecionar os dados do computador de bordo durante a operação, o que não é realizável ao implementar diretamente no *Titanium*, onde seria necessário interromper a atividade para acessar uma tela com os dados. No entanto, por esse mesmo motivo, uma desvantagem de se implementar na *Web* ou *mobile* é a necessidade de outro dispositivo

adicional para ter acesso à essas funcionalidades.

Como um ponto importante para o sucesso da ferramenta é a qualidade da interface, ou seja, mostrar os dados de forma clara e intuitiva, acredita-se que a flexibilidade e a quantidade de recursos gráficos da plataforma sejam relevantes para a análise. Nesse quesito, as plataformas *Web* e *mobile* são consideradas mais adequadas se comparadas com a interface embarcada.

Os critérios de usabilidade e agilidade de desenvolvimento são relativamente subjetivos, porém acredita-se que a melhor opção seja a plataforma *mobile*, pois as interfaces de usuário dos aplicativos atualmente seguem alguns padrões entre si, o que facilita a sua utilização por novos usuários. Quanto ao desenvolvimento do código, acredita-se que tanto *Web* quanto *mobile* teriam vantagem em comparação com o desenvolvimento somente embarcado, pois não se tem nenhuma familiaridade em programação utilizando a linguagem *C++*. Além disso, teria que seguir os padrões de programação já consolidados por parte da equipe de desenvolvimento embarcado.

De modo geral, as plataformas *Web* e *mobile* se mostraram bastante similares quanto aos critérios explorados. Portanto, acredita-se que, além de permitir uma expansão do escopo do projeto no futuro, desenvolver a aplicação para dispositivos móveis terá maior alcance de usuários, visto que estes são mais portáteis e, relativamente, mais utilizados no dia-a-dia. Pode-se argumentar que é possível acessar a aplicação *Web* através de um navegador no *smartphone*, porém, como explicado na seção 3.1, este não permite a utilização de alguns recursos nativos do dispositivo. Além disso, aplicativos móveis possuem grande apelo comercial atualmente e podem ser utilizados para atrair clientes novos a fechar parcerias com a Hexagon.

Uma tabela com os critérios de forma resumida é dada na Tabela 1.

Tabela 1 – Comparação entre as possíveis plataformas de desenvolvimento.

Critério de comparação	*Titanium*	*Web*	*Mobile*
Agilidade de desenvolvimento	Devagar	**Regular**	**Regular**
Usabilidade	Ruim	**Boa**	**Boa**
Interface de usuário	Rígida	**Flexível**	**Flexível**
Suporte à conexão remota	Não	**Sim**	**Sim**
Suporte à conexão local	Sim	Não	**Sim**
Útil durante a operação	Não	**Sim**	**Sim**
Necessita equipamento e/ou *SW* externo	**Não**	Sim	Sim
Suporte à múltiplos *displays*	Não	**Sim**	**Sim**
Inspeção simultânea entre usuários	Não	**Sim**	**Sim**
Alcance	Regular	Regular	**Alto**

Fonte – Arquivo pessoal.

4.5 PESQUISA DE MERCADO

Dada a escolha da plataforma móvel, a fim de identificar o cenário atual de aplicativos móveis, fez-se uma pesquisa de mercado de aplicativos das empresas concorrentes. Foram escolhidas empresas que produzem computadores de bordo para a agricultura e que também implementam as soluções de *software* dentro dele. Pesquisou-se na loja de aplicativos para dispositivos *Android* (*Play Store*) todos os aplicativos publicados pelas empresas concorrentes.

Algumas empresas, como a *Trimble Inc.* e *John Deere* possuem uma grande quantidade de aplicativos publicados, mas que possuem baixo número de instalações e escassas avaliações. Do lado nacional, no momento em que se fez a pesquisa, a empresa *Solinftec* possui 2 aplicativos publicados, sendo que um havia sido recentemente incluído na loja, com somente dezenas de instalações. No contexto do projeto desenvolvido, encontrou-se apenas aplicativos que fazem a transmissão da tela do computador de bordo e permitem o controle de forma remota, similar ao sistema de acesso remoto da Hexagon apresentado na seção 2.2.

Quanto aos que foram publicados pela Hexagon corporativa e suas outras divisões, encontrou-se poucas dezenas de aplicativos para diversos contextos de aplicação e com um número bem reduzido de *downloads*. Já para a divisão de agricultura da Hexagon, foi publicado apenas uma aplicação para acesso ao sistema *AgrOn Sala de Controle*, o qual é desenvolvido e mantido pela unidade de Ribeirão Preto da Hexagon.

Portanto, pode-se concluir que, até o momento, a Hexagon não fez grandes investimentos no desenvolvimento de aplicativos para serem utilizados em conjunto com o computador de bordo. Apesar das dificuldades de manter aplicações *mobile*, acredita-se que criar uma aplicação que irá atuar em conjunto com o computador de bordo, a fim de melhorar a qualidade dos produtos ao identificar mais rapidamente os problemas, pode ser um bom ponto de partida para o desenvolvimento de produtos *mobile* na empresa.

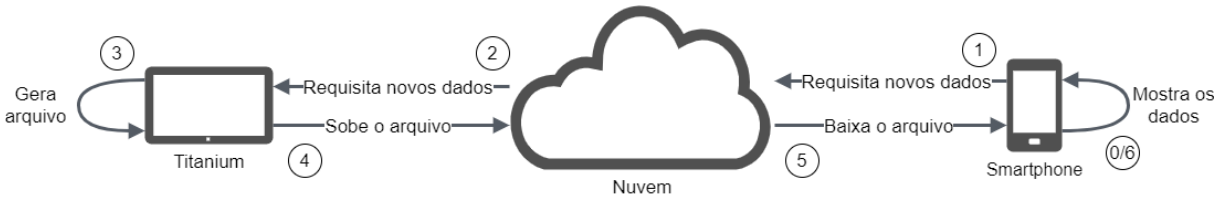
4.6 ARQUITETURA GERAL DO SISTEMA

Sob uma ótica em mais alto nível, a arquitetura do sistema pode ser dividida em três subsistemas: o subsistema embarcado (*Titanium*), o subsistema móvel (aplicativo para *smartphones*) e o subsistema em nuvem. Sendo assim, o primeiro é responsável pela coleta dos dados, processamento e organização em um arquivo; o segundo é responsável pela interação com o usuário e apresentação dos dados de forma gráfica; e o último pela transmissão e armazenamento dos dados em nuvem.

Quanto ao fluxo de dados do sistema, esse é composto de ciclos de envio e recebimento dos dados de saúde do computador de bordo, sendo representado pela Figura 12 em forma de etapas simbolizado pelas setas numeradas.

O ciclo se inicia com os últimos dados referentes ao último arquivo recebido sendo apresentados na tela do *smartphone* do usuário (passo 0), o qual pode requisitar novos dados através da nuvem (passo 1). Essa, por sua vez, encaminha a requisição para o devido computador de bordo (passo 2), onde são coletadas as novas informações e agrupadas em um arquivo de inspeção (passo 3). O arquivo novo é enviado para a nuvem (*upload*) por meio do serviço do *Cloud Sync* (passo 4). Chegando na nuvem esse arquivo fica armazenado, permitindo o *download* por algum usuário interessado no dado computador de bordo fonte (passo 5). Ao realizar o *download* de um arquivo de inspeção novo, o aplicativo irá atualizar a interface de acordo (passo 6).

Figura 12 – Ciclo de dados resumido do *TiX Inspector*



Fonte: Arquivo pessoal.

Os detalhes de modelagem e implementação de cada um desses subsistemas são discutidos separadamente nos capítulos a seguir.

5 APLICATIVO *MOBILE*

Dentro do sistema do *TiX Inspector*, o aplicativo *mobile* possui o papel de interagir com o usuário, apresentando os dados de inspeção da melhor forma possível e provendo algumas funcionalidades. Com isso, pretende-se auxiliar no diagnóstico de problemas e melhorar a comunicação entre as partes, buscando garantir o cumprimento dos objetivos definidos anteriormente.

De forma geral, o fluxo completo de utilização do aplicativo se inicia com o usuário inserindo os dados de *login* e se autenticando com o serviço da nuvem. Com isso, ele tem acesso ao aplicativo de fato, onde irá selecionar o computador de bordo que deseja inspecionar e verificará se este enviou dados recentemente. Então, requisitará novos dados e, ao chegarem, visualizará os dados que julgar interessante, os quais poderá favoritar se desejar. Caso algum dado esteja suspeito, o usuário pode gerar um PDF com os dados correntes, o qual pode ser compartilhado com outras pessoas e/ou visualizado em outro momento.

Nesse capítulo, serão discutidos os assuntos acerca da modelagem e implementação do aplicativo *mobile* do *TiX Inspector*, comentando sobre os pontos mais importantes, ilustrando através de diagramas e exemplos, quando possível. Vale ressaltar que o subsistema *mobile* foi totalmente especificado, projetado e implementado pelo autor deste documento, contando com a ajuda de alguns colaboradores da empresa acerca da usabilidade e *design* do aplicativo.

5.1 PROTÓTIPOS DE TELAS

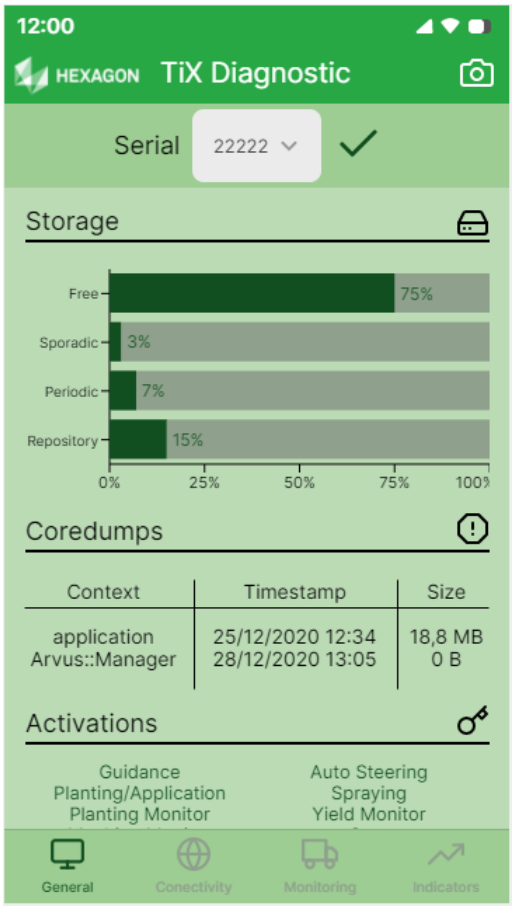
Antes de serem programadas as telas do aplicativo, criaram-se protótipos interativos utilizando uma ferramenta *Web* chamada *Framer* (B.V., c2021). Dessa forma, foi possível expor as ideias do projeto de maneira mais clara, podendo debater alguns pontos com os colegas e gestores com base na navegação do protótipo. Além disso, um bom protótipo navegável acaba reduzindo o tempo de implementação, pois este já define grande parte do que será o aplicativo (telas, botões, navegação etc.).

Em um primeiro momento, definiu-se que os contextos de dados estariam separados a partir de abas na barra inferior da tela, algo recorrente na maioria dos grandes aplicativos móveis atuais, como *Instagram*, *Spotify*, *Youtube* etc. Cada aba é composta por diversas seções, as quais são relacionadas com os dados que são mostrados, como, por exemplo, a seção *Coredumps* possui uma tabela com a lista de *core dumps* presentes no computador de bordo. O formato gráfico dos dados serão chamados de *widgets* e pretende-se utilizá-los em três tipos: tabelas, listas (com rótulo/*label* e valor) e gráficos.

Quanto às outras partes da interface, adicionou-se um cabeçalho com a logo da empresa, o nome do aplicativo e um botão para que dispare a ação de gerar um

arquivo PDF com os dados atuais. Ao clicar no botão, o usuário será perguntado se tem certeza de sua ação em uma caixa de diálogo. Um exemplo de tela da primeira versão do protótipo pode ser vista na Figura 13.

Figura 13 – Tela “Geral” no protótipo inicial.



Fonte: Arquivo pessoal.

No entanto, percebeu-se que esse modelo de interface não teria a capacidade de comportar uma quantidade grande de contextos (abas), limitando a solução como um todo. Além disso, o protótipo não cumpria o requisito de permitir a adição de seções favoritas no formato em que estava.

Portanto, decidiu-se criar um novo protótipo, no qual seria mantido o padrão de abas na parte inferior da tela, porém com um significado mais geral. Ou seja, dividiu-se a interface em página inicial, favoritos e perfil. Assim, a página inicial é constituída por uma lista de botões que levam para as telas de cada contexto (chamados de módulos), podendo ser expandida indefinidamente.

Além disso, adicionou-se uma barra onde é mostrado o número de série do computador de bordo escolhido e se este está disponível ou não. Ao clicar no botão do número de série escolhido, um modal irá aparecer para que o usuário escolha um novo

número de série. Já o botão de disponibilidade, durante a fase de desenvolvimento, foi alterado para mostrar o dia e hora dos últimos dados recebidos na nuvem. Com isso, o usuário pode ter uma noção da disponibilidade do computador de bordo.

Outro ponto positivo foi a adição de um botão de coração em cada seção para marcá-la como favorito e a aba de favoritos, onde essas seções podem ser conferidas rapidamente. Já na aba de perfil, estabeleceu-se que ela terá alguns botões que levam para outras telas mais simples, como uma tela “Sobre”, com a finalidade de mostrar algumas informações sobre o aplicativo. Ademais, tem-se uma tela em que serão listados os arquivos PDF já exportados, os quais podem ser visualizados, excluídos ou compartilhados.

Na Figura 14a é apresentado a tela do módulo “Geral” para fins de comparação com o primeiro protótipo, e, na Figura 14b, apresenta-se a página inicial, na qual foram criados dois estilos de botão dos módulos para posterior escolha.

Os protótipos completos com todas as telas podem ser visualizados no Apêndice A.

5.2 MODELAGEM DO APLICATIVO

De posse do protótipo das telas, prosseguiu-se para a modelagem do aplicativo, sendo a parte dos elementos visuais tratados pelos componentes *React Native* (abordados nas seções 5.3 e 5.4) e os elementos relacionados ao estado do aplicativo tratados pelos componentes *Redux* (abordado na seção 5.5). Além disso, definiu-se alguns módulos que irão realizar procedimentos comuns que não estão relacionados com a parte gráfica nem o estado, como manipular os dados a serem salvos no armazenamento local, por exemplo.

5.3 COMPONENTES VISUAIS

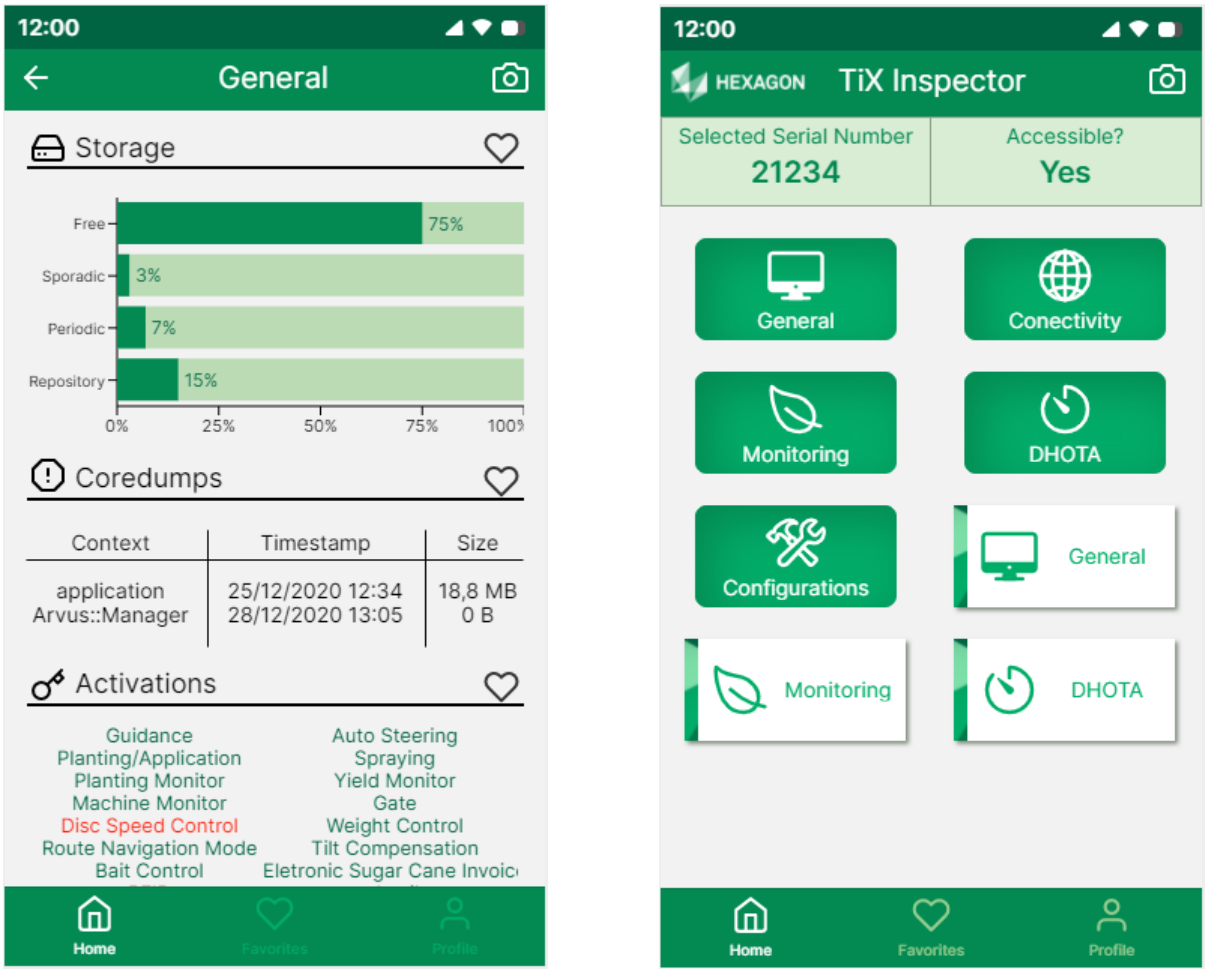
Os componentes do *React* são construídos a partir da composição de outros componentes nativos ou de bibliotecas da comunidade, os quais são parametrizados e customizados de acordo com o comportamento desejado.

Utilizando o protótipo final desenvolvido anteriormente, fez-se o trabalho de organizar os componentes para que esses tivessem a disposição definida. Através de uma árvore de componentes é possível descrever toda a estrutura visual do aplicativo, no qual o componente raiz é o “*App*”, como pode ser visualizado na Figura 16.

A partir do componente raiz, adicionou-se componentes de navegação (explicados com mais detalhes na seção 5.4), os quais são compostos de todas as opções de componentes (telas) disponíveis para o usuário navegar.

Quanto a página inicial do aplicativo, esta é composta por um cabeçalho (chamado de “*Header*”), uma barra contendo a informação do computador de bordo a ser

Figura 14 – Telas do protótipo final



(a) Tela “Geral” no protótipo final.

(b) Página inicial no protótipo final.

Fonte: Arquivo pessoal.

inspecionado (“*Serial Information*”) e a lista de botões que levam para as telas de cada contexto. Esses contextos são chamados de módulos e cada um deles tem uma tela correspondente no aplicativo contendo os dados relacionados à ele.

Os componentes que representam as telas dos módulos foram desenvolvidos de forma genérica a fim de serem reutilizados em todos os módulos, mudando seu conteúdo de acordo com a propriedade passada em sua construção. Cada módulo (“*Module*”), apresentado pelo componente “*Base Screen*”, é composto por um conjunto de seções (“*Sections*”), as quais possuem um cabeçalho (“*Section Header*”) e um conjunto de “*Widgets*”, que podem ser dos tipos lista (“*List*”), tabelas (“*Table*”) ou padrão (“*Default*”). O tipo do *widget* é escolhido levando em consideração a melhor forma de apresentar os dados e é definido em sua criação no estado inicial da aplicação, o qual será explorado na seção 5.5.1.

Sobre a tela de favoritos, é feita a renderização condicional baseada na quantidade de *widgets* favoritados pelo usuário. Caso possua valor zero, é apresentado

uma tela simples com uma mensagem o avisando de que não há nenhum favorito no momento. Caso contrário, apresenta-se uma tela com os *widgets* marcados como favoritos, de maneira idêntica à tela dos módulos, pois o mesmo componente "*Base Screen*" é utilizado.

A tela de *login* consiste em um encadeamento de componentes nativos com o objetivo de replicar a tela de *login* existente no sistema AgrOn Sala de Controle. Apenas por questões visuais, julgou-se necessário criar um componente para a área de texto de usuário e senha, a fim de adicionar um ícone à direita. Já a tela de perfil ("*Profile*") é construída com um cabeçalho grande ("*Header Big*") e uma lista de botões ("*Profile Button*").

Alguns dos componentes criados são detalhados separadamente na Figura 16 para simplificar a apresentação no diagrama geral da Figura 15. O componente mais complexo desses é o "*Serial Information*", contendo os dois botões definidos no protótipo e um modal. Este é composto especialmente por um "*Picker*", que apresenta a lista de números de série que podem ser escolhidos ("*Picker Item*"), e um botão para confirmar a seleção.

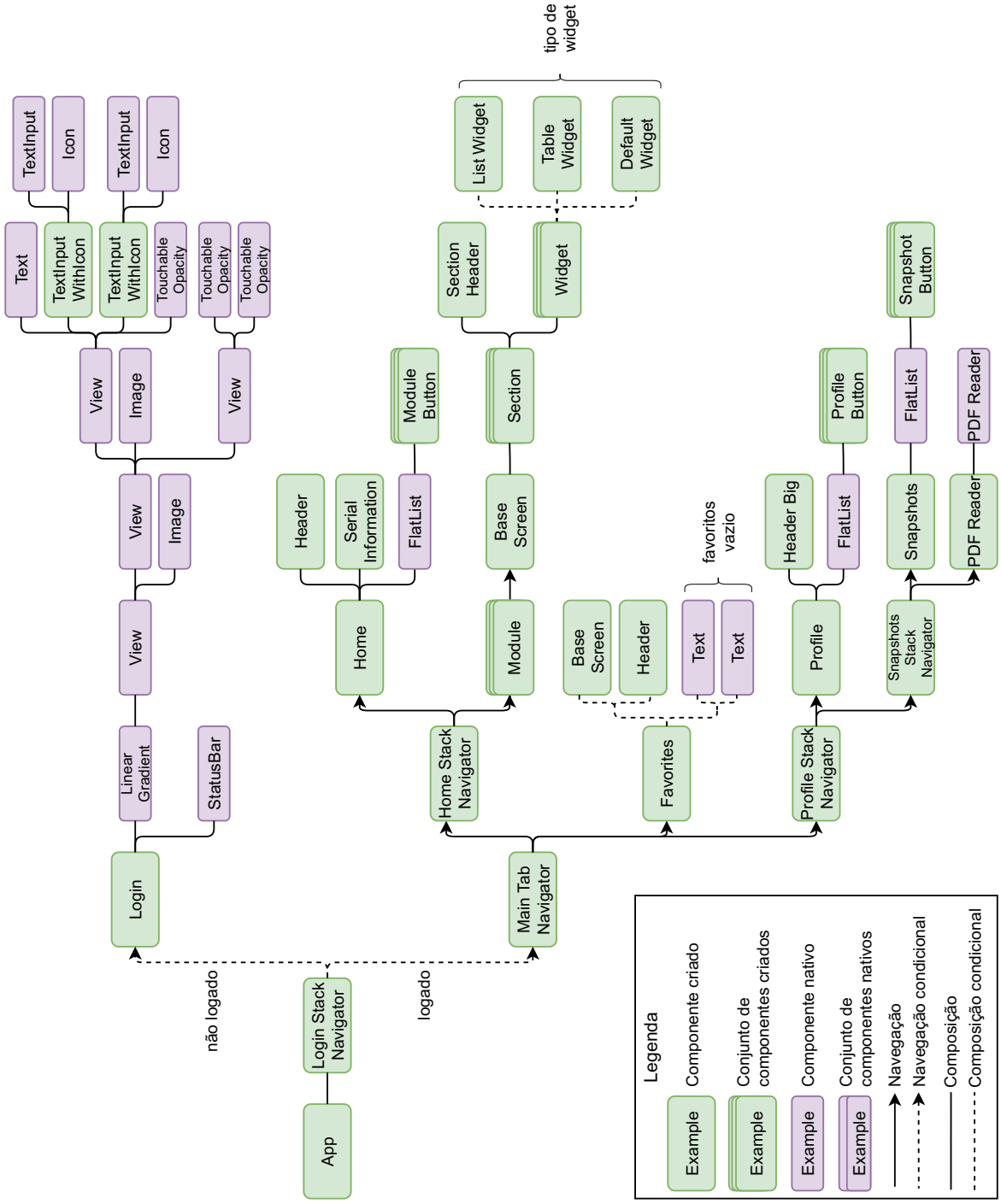
Outros componentes importantes são os *widgets*, sendo o componente do tipo "lista" basicamente uma lista de textos dispostos em duas colunas, na qual a primeira representa o *label* e a segunda os valores; o componente de tabela composto por um título e um componente de tabela dado por um componente "*Row*" como cabeçalho e um componente "*Rows*" como as linhas; e o componente padrão contendo apenas um texto, utilizado quando houve algum erro de carregamento do *widget*.

5.4 NAVEGAÇÃO

A navegação no aplicativo foi organizada utilizando os tipos de navegadores *Stack* e *Tab navigators*. Como o nome sugere, o navegador *Stack* age de forma a empilhar as telas em cima da tela anterior, mantendo o histórico do caminho feito pelas telas nessa pilha, permitindo que o usuário volte para a tela anterior. Possui comportamento similar as páginas da *Web* e a forma como o usuário navega pelas telas nos *sites*. Já o navegador *Tab* organiza as telas em abas, utilizando um conceito mais horizontal de navegação, sem hierarquia entre elas. Com isso, o usuário consegue percorrer as telas sem precisar voltar e consegue trocar de aba independente de qual esteja no momento.

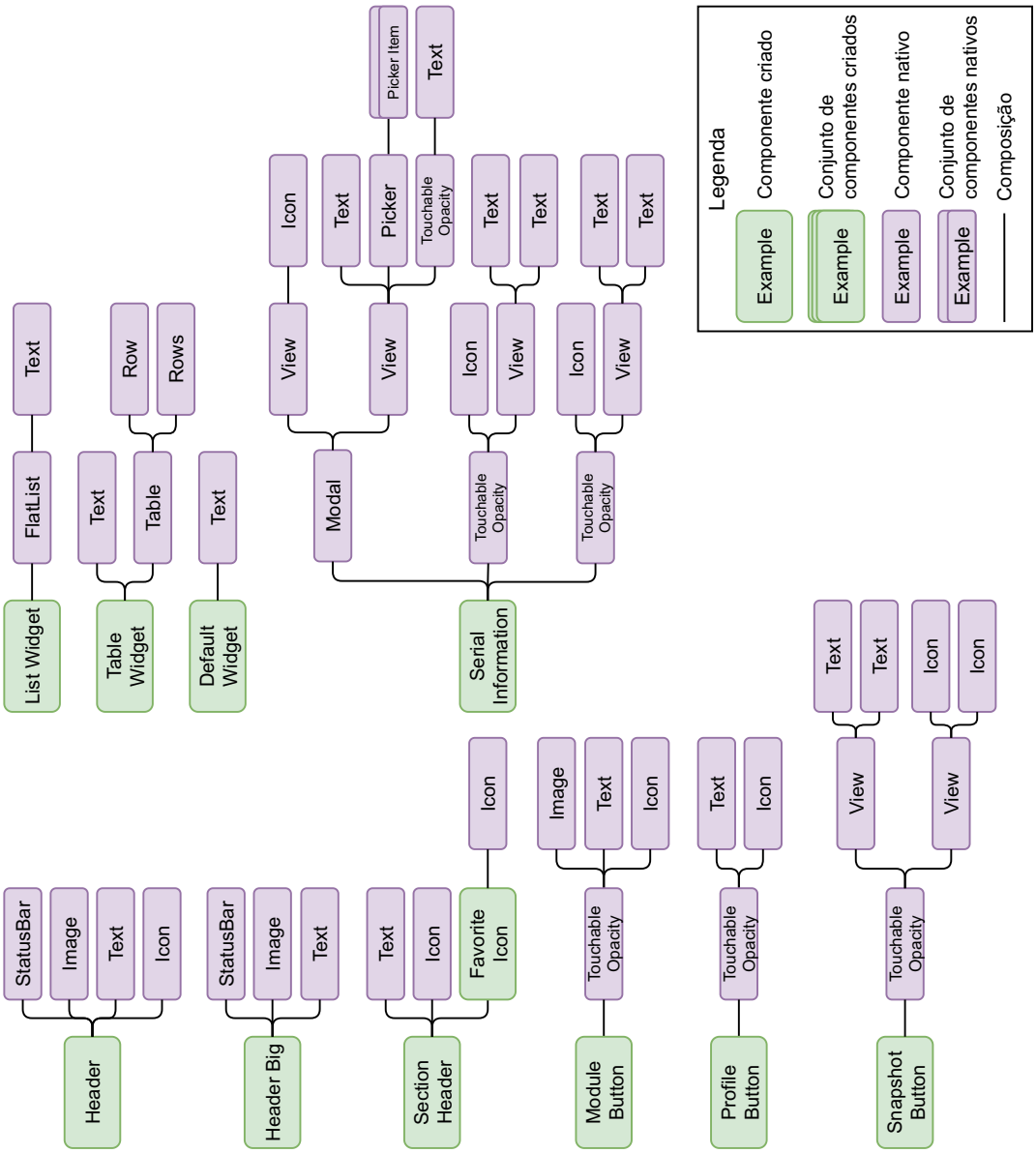
Ao abrir o aplicativo pela primeira vez, o usuário recebe a tela de *login* para que possa passar seu usuário e senha do sistema. Com os parâmetros validados, ele prossegue para a página inicial do aplicativo. Para permitir esse comportamento, utilizou-se o navegador *Stack* ("*Login Stack Navigator*", na Figura 15), em que a tela inicial é a tela de *login*, caso o usuário não tenha *logado* até o momento. Caso contrário, a tela principal é mostrada.

Figura 15 – Diagrama geral dos componentes.



Fonte: Arquivo pessoal.

Figura 16 – Diagrama dos componentes criados.



Fonte: Arquivo pessoal.

Para a parte principal do aplicativo, utilizou-se o navegador *Tab* ("*Main Tab Navigator*") contendo as abas "*Home*", "*Favorites*" e "*Profile*", pois os contextos dessas telas são independentes entre si.

Na parte de navegação do usuário para visualizar os dados, fez-se uso do navegador *Stack* ("*Home Stack Navigator*"), contendo a tela inicial ("*Home*") e as telas dos módulos. Assim, os botões da tela inicial tem o papel de redirecionar para as outras telas da pilha, sendo necessário voltar à tela inicial, caso o usuário queira visualizar os dados de outro módulo.

Um comportamento semelhante é utilizado na aba de perfil, na qual cada um dos botões leva para uma tela diferente. Portanto, o "*Profile Stack*" contém a tela geral de perfil ("*Profile*") e as outras subtelas. Porém, houve a necessidade de criar um novo navegador *Stack* para a tela de *Snapshots*, pois é permitido que seja visualizado o arquivo PDF em um outra tela dentro da tela "*Snapshots*". Dessa forma, o "*Snapshots Stack*" possui a tela "*Snapshots*" com a lista de arquivos (apresentados pelos componentes "*Snapshot Button*") e a tela "*PDF Reader*" para leitura do arquivo.

5.5 *REDUX* NO *TIX INSPECTOR*

Como explicado na seção 3.4, a arquitetura *Flux* traz diversas vantagens quanto a implementação de aplicações utilizando *React*, como robustez, escalabilidade, entre outros. Nessa seção, serão tratados os conceitos de estado, ações, *reducers* e *selectors* no âmbito do *TiX Inspector*, dando exemplos e explicando as especificidades da implementação escolhida.

Vale comentar que, em uma implementação mais recente do *Redux*, foi adicionado um conceito de partições do *store*, chamadas de *slices*. Apesar de possuir várias partes, a ideia de *store* único ainda é mantida, pois todas elas são fundidas na sua criação. Mais detalhes de como os *slices* funcionam serão dados em seguida, com foco em cada um dos elementos da arquitetura.

5.5.1 O estado

De acordo com a biblioteca *Redux*, o estado tem como objetivo armazenar os dados que são compartilhados entre os componentes da interface, sendo a única fonte do estado das variáveis do programa.

Utilizando o conceito de *slices*, dividiu-se o estado do aplicativo nas seguintes partes: módulos, seções, *widgets*, *displays*, *login* e *snapshots*. A estrutura simplificada do estado pode ser visto na Figura 17.

A estrutura e organização da interface do *TiX Inspector* é definida pelos *slices* de módulos, seções e *widgets*, o qual possui uma chave para cada tipo de *widget*, cujo valor é uma lista de objetos, e o *timestamp* da última atualização. Como explicado

anteriormente, há uma hierarquia entre essas entidades, sendo que os módulos são a entidade mais abrangente.

Cada uma das entidades é representada por um objeto *Javascript* onde possuem uma propriedade identificadora (ID). É por esse identificador que são feitas as relações de composição das entidades entre si. Quanto aos módulos em específico, esses possuem apenas o ID, nome e o nome do ícone que estará no seu botão localizado na tela "*Home*".

As seções possuem um ID, nome do ícone e título (localizados no cabeçalho da seção), uma *flag* para indicar se está favoritado, o ID do módulo ao qual pertence e os dados que estão contidos nela. Os dados da seção são indicados por uma lista de objetos, os quais possuem o tipo e o ID do *widget*, e, opcionalmente, se a informação será apresentada com uma barra de rolagem e sua altura na tela. De forma resumida, as seções definem em qual módulo estarão localizadas e quais os *widgets* que pertencem a ela.

Procurou-se desacoplar configurações de como o *widget* será apresentado e em qual(is) seção(ões) esse pertence a fim de deixar a cargo da configuração de cada seção customizar o *widget* da forma que for conveniente para tal. Além disso, preferiu-se deixar a organização das entidades da interface o mais customizável possível, para que não haja problemas de restrição do aplicativo no futuro.

Além dessas informações, armazena-se no estado a lista de computadores de bordo passíveis de inspeção, qual deles está selecionado para inspeção, o *timestamp* do último pacote recebido dele, uma lista de nomes dos *snapshots* guardados no *smartphone*, e informações acerca do *login*.

Quanto a esse último, tem-se armazenado uma *flag* para indicar se está *logado*, usuário e senha definidos pelo usuário na tela de *login* e as seguintes informações adquiridas ao fim da autenticação: nome do usuário, empresa, grupos aos quais pertence e os *tokens* para autenticação das chamadas à API do *TiX Inspector* na AWS.

Os exemplos das estruturas específicas do estado, como o formato das listas e seções, por exemplo, podem ser vistas no Apêndice B.

5.5.2 Ações e *Reducers*

Viu-se que, na arquitetura *Flux*, a mudança dos estados é dada de acordo com o disparo de ações pelas *Views* e pelo usuário. Então, definiu-se que as ações síncronas permitidas para cada *slice* do aplicativo são:

- Seções: alternar *flag* de favorito.
- *Widgets*: atualizar *widget*, alternar estado de carregamento e definir último *timestamp* recebido.

Figura 17 – Estrutura básica do estado do aplicativo.

```
let store = {
    widgets: {
        tables: [],
        lists: [],
        graphs: [],
        lastTimestampReceived: '',
    },
    modules: [],
    sections: [],
    displays: {
        registered: [],
        selected: 0,
    },
    login: {
        isLogged: true,
        username: '',
        password: '',
        name: '',
        company: '',
        groups: [],
        tokens:  {
            accessToken: '',
            idToken: '',
            refreshToken: '',
        }
    },
    snapshots: [],
}

```

Fonte: Arquivo pessoal.

- *Displays*: adicionar e remover número de série da lista de computadores de bordo disponíveis, e trocar o que está selecionado para inspeção.
- *Login*: realizar *login* e *logout*.
- Módulos e *Snapshots*: nenhuma ação.

Há casos onde é necessário resgatar dados através de funções assíncronas, como baixar os arquivos de inspeção novos, por exemplo. Portanto, as ações assíncronas disponibilizadas são:

- *Widgets*: requisitar novo pacote, consultar quais são os novos pacotes, o horário do pacote mais recente e baixar os arquivos novos.
- *Snapshots*: atualizar lista de *snapshots*.
- *Displays*: atualizar os números de série registrados na nuvem para o usuário.

Como comentado na seção 3.4, os *reducers* implementam as lógicas de mudança no estado da aplicação baseado na ação e no estado atual. Com a utilização do conceito de *slices*, os *reducers* são organizados de acordo com a parte do estado que irão alterar, ou seja, uma função que altera o valor de um *widget* deve ficar no *slice* de *widgets*, por exemplo.

Teoricamente, os *reducers* devem ser implementados de maneira que sejam funções puras. No entanto, com o intuito de facilitar o desenvolvimento, a biblioteca de implementação do *Redux* permite a mutação do estado nos *reducers*, transformando-os em funções puras internamente.

Uma outra facilidade de se implementar utilizando essa biblioteca é que, ao estabelecer os *slices*, as funções criadoras de ações (*Action creators*) são geradas automaticamente. Assim, para despachar uma ação ao *store*, passa-se apenas os parâmetros, caso estes sejam necessários.

Thunks são funções de ordem superior, as quais recebem os parâmetros da ação disparada e retornam uma função assíncrona. Essa função assíncrona pode se resolver com sucesso ou com falha, despachando outras ações de acordo.

Cada ação síncrona possui um *reducer* associado, já uma ação assíncrona é composta por um *thunk* que se resolve em diversos *reducers* de acordo com o resultado da função *thunk*. Nas Figuras 18 e 19, pode-se ver os exemplos de *reducers* síncronos e assíncronos, respectivamente, que foram implementados no aplicativo.

Quanto à função síncrona de atualizar o valor dos *widgets*, essa recebe uma ação cujo *payload* consiste em um objeto "*target*" com o tipo e ID do *widget* a ser atualizado, e o novo valor. Dentro do *reducer*, o estado é apenas a parte de sua responsabilidade, ou seja, apenas os *widgets*. No início da função, filtra-se o *widget* escolhido através do ID na lista dos *widgets* de mesmo tipo. Em seguida, conforme o tipo do *widget*, faz-se a atualização do valor utilizando o novo valor passado no *payload* da ação.

A respeito da função assíncrona, usou-se o método "*createAsyncThunk*", o qual recebe como parâmetros o nome do *reducer* associado e a função assíncrona. Esse método facilita a criação de *thunks*, sendo disponibilizado pela mesma biblioteca comentada anteriormente. Na Figura 19 cria-se um *thunk* para baixar os novos arquivos, passando como nome do *reducer* "*widgets/getNewFiles*" e a função assíncrona que irá acessar a API do subsistema em nuvem do *TiX Inspector*, passando os parâmetros necessários. Caso tenha sucesso, o *thunk* irá despachar uma ação para o *reducer* "*getNewFiles.fulfilled*", o qual irá gravar no estado o *timestamp* mais recente dos arquivos baixados.

Figura 18 – Exemplo de *reducer* síncrono.

```
update: (state, action) => {
    let type = action.payload.target.type;
    let selected = state[type].filter(({id}) => id === action.payload.target.id)[0];

    if (type === 'lists') {
        selected.values = action.payload.value !== undefined ? action.payload.value : [];
    } else if (type === 'tables') {
        selected.rows = action.payload.value !== undefined ? action.payload.value : [];
    }
}

```

Fonte: Arquivo pessoal.

Figura 19 – Exemplo de *reducer* assíncrono.

```
export const getNewFiles = createAsyncThunk(
    'widgets/getNewFiles',
    async ({ newPkgs, idToken }) => {
        newPkgsEncoded = encodeURI(JSON.stringify(newPkgs))
        const response = await fetch(
            'URL' + '?newPkgs=' + newPkgsEncoded,
            {
                method: 'GET',
                headers: {
                    'Authorization': idToken
                }
            }
        )
        return JSON.stringify(await response.json())
    }
)

[getNewFiles.fulfilled]: (state, action) => {
    allInfoTimestamps = Object.keys(JSON.parse(action.payload)["inspection.json"])
    maxTimestamp = Math.max.apply( elem => elem.parseInt(), allInfoTimestamps).toString()
    state.lastTimestampReceived = maxTimestamp
}
```

Fonte: Arquivo pessoal.

5.5.3 *Selectors*

Assim como a escrita no estado *Redux* é permitida apenas por meio dos *reducers*, a leitura é permitida apenas através dos *Selectors*. Estes são funções que recebem, na entrada, o estado completo da aplicação e retornam (selecionam) a parte desejada. Porém, essas funções só funcionam corretamente se forem passadas como argumento do método "*useSelector*" da API do *store*.

Um exemplo de *selector* é apresentado na Figura 20, onde ele recebe o estado completo, acessa a chave das seções ("*sections*") e filtra as seções que possuem a propriedade "*isFavorite*" como verdadeira.

Figura 20 – Exemplo de *selector*.

```
export const getFavoriteSections = state =>
    state.sections.filter(section =>
        section.isFavorite
    )

```

Fonte: Arquivo pessoal.

5.5.4 Entidade *linker*

Criou-se uma entidade chamada *linker* a fim de mapear os dados contidos no arquivo JSON de inspeção para os *widgets*. Utilizou-se um JSON com uma lista de objetos seguindo uma estrutura de entrada e saída ("*from*" e "*to*", respectivamente) similar a uma rota que o dado irá seguir. Ou seja, na chave "*from*", define-se o caminho da chave no JSON de inspeção e, na chave "*to*", escolhe-se o *widget* para apresentar o dado. Na Figura 21, pode-se verificar um exemplo de mapeamento dos dados no *linker*.

Implementou-se que, ao chegar dados novos, são percorridas todas as rotas, resgatado, no arquivo de inspeção, o dado relativo ao caminho definido na chave "*from*" e despachado uma ação para o *store* a fim de atualizar o *widget* definido na chave "*to*".

Percebeu-se que essa estrutura de rotas garantiu escalabilidade rápida ao se acrescentar novos dados na interface, visto que a estrutura é simples e repetitiva.

Figura 21 – Exemplo de rota do *linker*.

```
[
    {
        "from": "chave_externa/chave_do_dado",
        "to": {
            "type": "lists",
            "id": 0
        }
    },
    // ...
]

```

Fonte: Arquivo pessoal.

5.6 ARMAZENAMENTO LOCAL

Devido ao fato do estado ser perdido em cada inicialização do aplicativo, viu-se a necessidade de armazenar localmente, em memória não volátil, informações que devem ser persistidas entre as execuções. Julgou-se importante guardar os dados de *login* com o intuito de o usuário não precisar informar seu usuário e senha toda vez que usar o aplicativo. Ademais, decidiu-se manter o último computador de bordo escolhido para inspeção por motivos de comodidade na usabilidade.

Além de persistir os dados entre inicializações, viu-se a necessidade de armazenar os últimos dados recebidos da nuvem e seu *timestamp* para cada computador de bordo, pois, ao alterar qual será inspecionado, é possível resgatar os últimos dados do computador de bordo escolhido.

Quanto à implementação, dispôs-se de uma biblioteca da comunidade que faz o armazenamento seguindo o modelo de chave-valor. Dessa forma, criou-se um arquivo *Javascript* que exporta os métodos para gravação e leitura dos dados no armazenamento local. Para os dados que devem ser guardados para cada computador de bordo, definiu-se a chave como sendo o dado concatenado do número de série, como, por exemplo, "*timestamp12345*" para guardar o *timestamp* do último arquivo recebido para o número de série 12345.

Além dos dados do estado, conservam-se os arquivos PDF de *snapshot* no armazenamento local para posterior leitura ou compartilhamento.

5.7 *SNAPSHOTS*

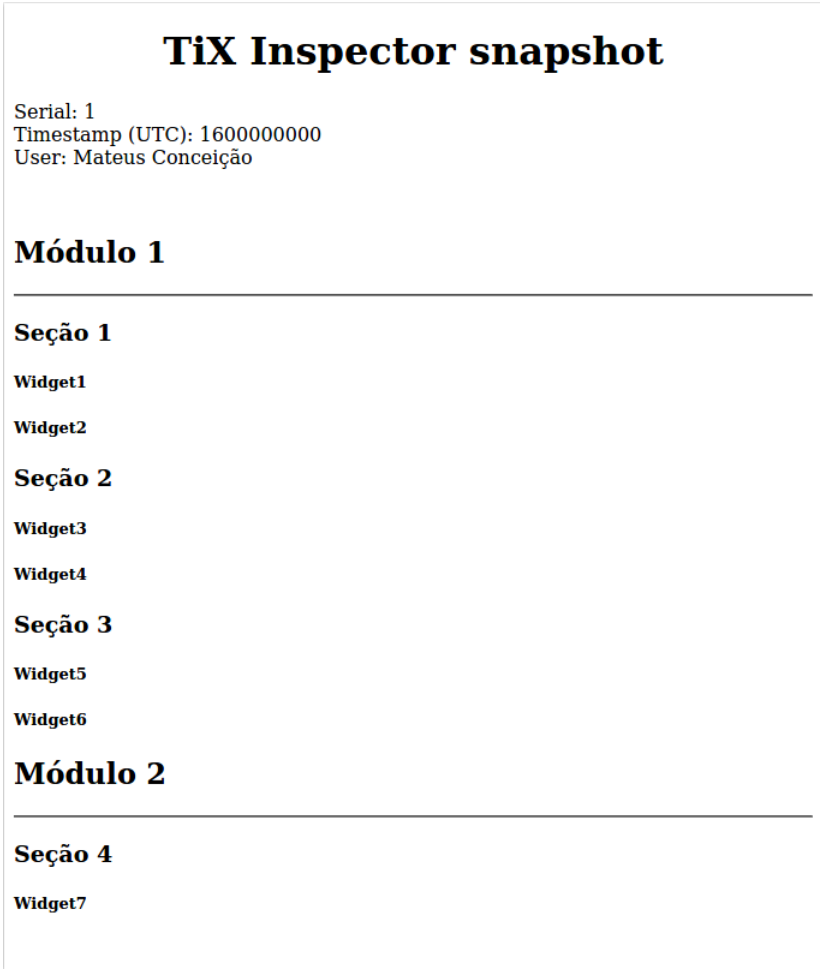
Como comentado anteriormente, julga-se útil desenvolver uma funcionalidade com o intuito de o usuário conseguir exportar os dados atuais da interface para um arquivo, como uma espécie de foto do estado do computador de bordo, e poder compartilhar e consultar em outro momento.

Para tal, utilizou-se de uma biblioteca que realiza a impressão para PDF de uma página HTML. Portanto, foi preciso transformar o estado do aplicativo no formato HTML, cujo formato é dado pelo modelo apresentado na Figura 22. Os arquivos PDF gerados possuem nome seguindo o formato "*SN_snapshot_TIMESTAMP.pdf*".

Utilizou-se o estado corrente no momento da requisição do *snapshot* como entrada dos métodos de conversão para HTML desenvolvidos. Com o intuito de gerar o HTML total, utilizou-se de uma composição de funções, ou seja, para gerar o HTML dos módulos, precisa-se do HTML das seções pertencentes a esse módulo, e assim por diante.

Para exemplificar a forma de criação do HTML, pode-se visualizar a Figura 23, na qual é mostrada a função "*createModuleHTML*", que recebe como entrada o objeto "*module*" e o estado, e retorna o HTML relativo à esse módulo. Em um primeiro

Figura 22 – Exemplo de PDF do *snapshot* exportado.



TiX Inspector snapshot

Serial: 1
Timestamp (UTC): 1600000000
User: Mateus Conceição

Módulo 1

Seção 1

Widget1

Widget2

Seção 2

Widget3

Widget4

Seção 3

Widget5

Widget6

Módulo 2

Seção 4

Widget7

Fonte: Arquivo pessoal.

momento, filtra-se as seções pertencentes ao módulo com o método "*filter*", seguido do método "*map*", a fim de executar uma função em cada um dos elementos, e, por fim, concatena-se todos os elementos utilizando o método "*reduce*". O retorno é dado seguindo o modelo proposto para cada entidade da interface.

Todos as funções de geração do HTML, impressão do HTML em um PDF, seu salvamento no *smartphone*, consulta a lista de arquivos PDF armazenados e deleção de um determinado PDF, foram colocados em um arquivo *Javascript* que pode ser utilizado em qualquer lugar do código, similar a uma biblioteca.

5.8 *LOGIN*

Como determinado nos requisitos do sistema, o *login* utiliza a mesma *pool* de usuários do serviço de acesso remoto descrito na seção 2.2. Com isso, é possível ter controle sobre quais os computadores de bordo o usuário terá acesso. Por meio da

Figura 23 – Método para criação do HTML para determinado módulo.

```
const createModuleHTML = (module, state) => {
    let sectionsHTML =
        state.sections.filter(section => section.moduleId === module.id)
                      .map(section => createSectionHTML(section, state))
                      .reduce((x, y) => x + y, "")
    return `
        <h2>${module.name}</h2>
        <hr>
        <div>
            ${sectionsHTML}
        </div>
    `
}

```

Fonte: Arquivo pessoal.

autenticação do usuário, consegue-se informações do nome, qual empresa o usuário está cadastrado, em quais grupos pertence etc. Assim, a partir dos identificadores dos grupos que o usuário está contido, pode-se realizar uma requisição à nuvem com o intuito de resgatar quais os números de série são permitidos para tais grupos.

Para a implementação, utilizou-se a mesma biblioteca utilizada pelos outros sistemas *Web* da empresa que necessitam de autenticação dos usuários cadastrados no serviço do *Cognito* da AWS. Disponibilizou-se um método em um arquivo *Javascript* para autenticação do usuário e senha, o qual, caso tenha sucesso, despacha uma ação de *login* para o *store* e guarda os dados de *login* no *smartphone*.

6 SISTEMA EM NUVEM

De maneira geral, no contexto do *TiX Inspector*, o subsistema em nuvem atua como meio de transporte e armazenamento das informações dos computadores de bordo até o *smartphone* do colaborador de suporte da Hexagon. Dessa forma, possibilita a inspeção dos computadores de bordo em campo remotamente.

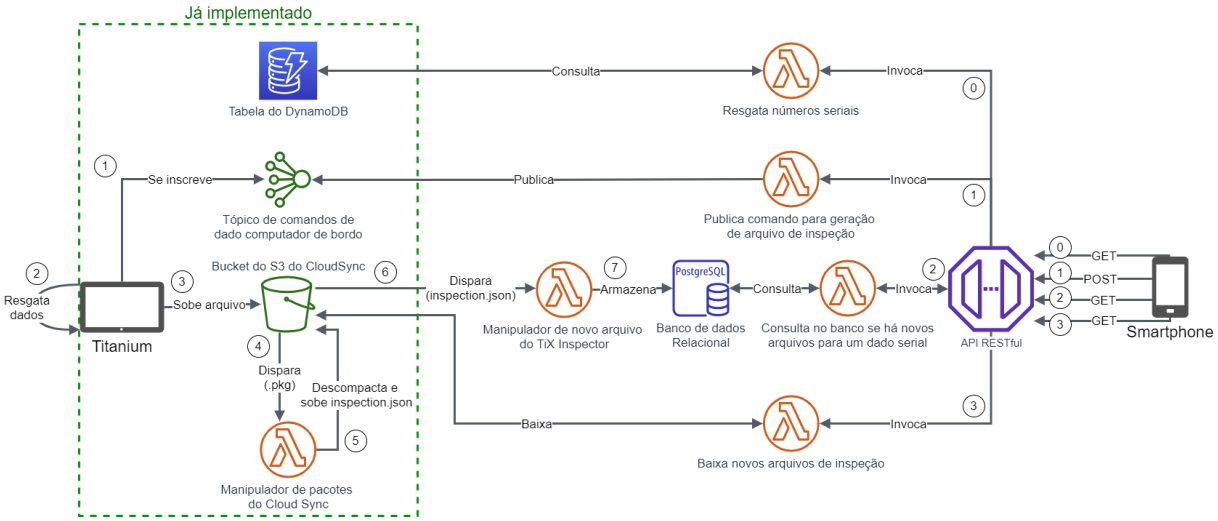
Como forma de ilustrar o funcionamento de todo o ciclo de dados do ponto de vista do subsistema em nuvem, fez-se a Figura 24, a qual mostra toda a estrutura desenvolvida em conjunto com o que já estava previamente implementado. A interface do subsistema em nuvem com o subsistema *mobile* é dada por meio de uma *API RESTful* (explicado com mais detalhes na seção 6.1). Já a interface com o computador de bordo é feita utilizando a infraestrutura já implementada pelos desenvolvedores da empresa, a qual era utilizada para outros fins e é descrita na seção 6.2.

Em um primeiro momento, o smartphone consulta em uma tabela do *DynamoDB* quais os números de série permitidos para o usuário. Com o número de série selecionado, o usuário envia, através do *smartphone*, uma requisição para o computador de bordo gerar um novo arquivo de inspeção com os dados atualizados. Essa requisição chega até o computador de bordo, o qual resgata os dados atualizados, compila em um arquivo de inspeção e envia para a nuvem por meio do serviço *Cloud Sync* interno a ele. Ao subir para a nuvem, informações acerca do arquivo em si são armazenadas em um banco de dados para posterior consulta. Quando o *smartphone* envia uma requisição para verificar quais são os arquivos novos, faz-se uma consulta no banco de dados, retornando os seus identificadores. De posse dos identificadores dos arquivos mais atuais, o *smartphone* solicita o conteúdo deles, o que desencadeia no *download* dos arquivos do *bucket* e retorno do conteúdo estruturado no formato esperado pelo aplicativo *mobile*.

Para a execução dos procedimentos de envio de mensagem *IoT*, processamento dos dados, consulta e escrita nos bancos de dados e *download* dos arquivos armazenados no S3, utilizaram-se funções *Lambda* dadas as suas vantagens e facilidades de implementação. Na seção 6.4, serão expostas a implementação de cada uma dessas funções.

Como representado pelo retângulo tracejado na Figura 24, há partes do subsistema em nuvem que já estão implementados atualmente, e que foram desenvolvidas pelos colaboradores do P&D da empresa. No entanto, o restante do subsistema foi desenvolvido pelo autor deste documento, seguindo as orientações do gestor de *Cloud* da Hexagon, a fim de seguir os padrões de implementação empregados no momento. Além disso, vale ressaltar que se implementou o subsistema em ambiente de desenvolvimento (*dev*) com o intuito de evitar problemas com os sistemas utilizados pelos clientes e permitir maior flexibilidade para mudanças.

Figura 24 – Estrutura do subsistema em nuvem para o *TiX Inspector*.



Fonte: Arquivo pessoal.

6.1 A API

O subsistema em nuvem possui quatro funcionalidades básicas que interagem com o subsistema *mobile*: verificar os números de série liberados para o usuário; requisitar um novo arquivo de inspeção; verificar arquivos novos para determinado computador de bordo a partir de certo momento; e realizar o *download* dos arquivos de inspeção definidos. Na Figura 24, essas funcionalidades são representadas pelas setas que partem do *smartphone* em direção à *API RESTful*, as quais são numeradas para indicar a sequência de sua execução.

Como explicado na seção 3.5.5, o serviço para implementação de interfaces *Web* na AWS é o *API Gateway*. Nele, as interfaces *RESTful* contém diversos métodos, os quais são organizados em recursos. Cada recurso pode ter um método de cada tipo (*GET*, *POST* etc.) e os recursos podem ser aninhados entre si.

Definiu-se que os recursos são:

- "*/displays*", para consulta dos números de série liberados para dado usuário através do método GET (método número 0 da Figura 24), cujo único parâmetro é a lista contendo os identificadores dos grupos aos quais o usuário pertence.
- "*/requests*", para requisição de um arquivo novo, contendo apenas um método POST (método número 1), o qual recebe o número de série do computador de interesse como parâmetro.
- "*/packages*", para consulta dos novos arquivos enviados, contendo apenas um método GET (método número 2). Esse método recebe o número de série e o

último *timestamp* recebido.

- "*/files*", representando os arquivos finais do ciclo e contendo apenas um método GET (método número 3), o qual recebe um JSON que define os arquivos a serem baixados.

Todos os métodos da API criada encaminham a chamada para uma função *Lambda* correspondente, a qual recebe os parâmetros e realiza os procedimentos necessários em outros serviços, retornando dados, se necessário.

Por exemplo, para invocar o método de número 1 para o computador de bordo de serial 1 a partir do timestamp 1600000000, deve-se definir como URL o caminho "*URLBase/packages?serialNumber=1&lastTimestamp=1600000000*", onde já estão incluídos os dois parâmetros necessários para execução do método ("*serialNumber*" e "*lastTimestamp*"). Além disso, precisa-se escolher o método GET e adicionar ao cabeçalho o autenticador (*ID Token*) na chave "*Authorization*".

Quanto à segurança das chamadas, por padrão, apenas invocações na API que contém, em seu cabeçalho, um *ID Token* válido são permitidas. Caso contrário, retorna-se um código de erro correspondente.

6.2 O SISTEMA ATUAL

Atualmente, tem-se reservado um tópico do serviço de *IoT* para envio de comandos ao computador de bordo, o qual é representado na Figura 24 como o tópico "Tópico de comandos de dado computador de bordo", em que existe um tópico específico para cada computador de bordo. Já está programado para o sistema do computador de bordo se inscrever nesse tópico quando este se conectar à nuvem (ação de número 1, na figura). Determinou-se que esse tópico de comandos será utilizado a fim de enviar o comando para geração de um novo arquivo de inspeção.

Como explicado na seção 3.7, o computador de bordo possui um serviço para *upload* de arquivos para a AWS chamado de *Cloud Sync*. Além disso, tem-se estrutura em nuvem para receber os arquivos enviados a um *bucket* do S3 específico (representado pelo "*bucket* do S3 do *Cloud Sync*"), o que dispara uma função *Lambda* ("Manipulador de pacotes do *Cloud Sync*"). Essa função é configurada para ser disparada apenas quando pacotes (arquivos .pkg) são recebidos e realiza a descompactação deles, subindo o seu conteúdo no mesmo *bucket*, mas em outra "pasta".

Ademais, como já comentado na seção 5.8, já existem tabelas implementadas no serviço *DynamoDB*, as quais armazenam e definem quais os números de série podem ser acessados pelos usuários. Essa relação é feita através de uma tabela de grupos, os quais possuem, dentre outros dados, uma lista de números de série pertencentes ao dado grupo. Em conjunto com essa tabela, cada usuário da *pool* de

usuários utilizada possui um atributo que determina o conjunto de grupos que ele pertence.

Com isso, o sistema atual já consegue receber comandos, enviar arquivos e deixá-los disponíveis para outras aplicações. Portanto, essas estruturas serão aproveitadas para implementação do projeto *TiX Inspector*.

6.3 O BANCO DE DADOS

Com o intuito de persistir as informações acerca dos arquivos de inspeção, decidiu-se implementar um banco de dados simples na plataforma AWS utilizando o serviço RDS.

Determinou-se que será armazenado apenas uma tabela composta pelas colunas "*id*", "*serial_number*", "*bucket*", "*file_key*" e "*file_timestamp*". Vale ressaltar que o S3 armazena os arquivos através de uma chave dividida por barras, não possuindo o conceito de organização por pastas. Dessa forma, basta armazenar a chave (*key*) do arquivo e o *bucket* a fim de encontrá-lo posteriormente.

Quanto à configuração, optou-se pelas configurações mais básicas possíveis com o propósito de reduzir ao máximo os custos com o projeto. Inclusive, considerou-se suficiente o plano sem custos visto que, por enquanto, o principal é validar o conceito do projeto. Com isso, o banco de dados fica *online* apenas durante um determinado período, em apenas uma região, sem replicações, com a capacidade de armazenamento mínima possível.

Uma parte das configurações diz respeito à rede privada e segurança do banco de dados. Para definir tais configurações, recorreu-se ao gestor da equipe de desenvolvimento em nuvem da empresa, o qual orientou a instalação do banco de dados em sub-redes dentro de uma das redes privadas. Assim, definiram-se regras para acesso limitado apenas às entidades de dentro das sub-redes. Portanto, seguiu-se as diretrizes de segurança da Hexagon e os padrões de implementação dos bancos de dados em nuvem.

6.4 AS FUNÇÕES *LAMBDA*

Para performar as operações que interligam os serviços, fez-se uso das funções *Lambda*, devido às facilidades e vantagens de se implementar dessa forma, como comentado na seção 3.5.3.

Escolheu-se a linguagem de programação *Python*, pois é a mais comum nas outras funções *Lambda* desenvolvidas pela equipe de nuvem da empresa, facilitando a manutenção e entendimento futuro dos códigos.

Pode-se configurar um evento de disparo para as funções *Lambda* advindo de alguns outros serviços da AWS. Portanto, para as funções ligadas à API, definiu-

se que a chamada dos métodos da API irão dispará-las. Já para a *Lambda* que irá adicionar um novo registro no banco de dados, ela será disparada por um novo arquivo "*inspection.json*" colocado no *bucket*.

Por padrão, cada função possui uma *Role*, a qual define quais recursos da AWS que a *Lambda* terá acesso. Assim sendo, para cada recurso adicional que a *Lambda* precisa acessar, como o *bucket* do *Cloud Sync*, por exemplo, acrescenta-se a devida permissão na *Role* da *Lambda*.

Quanto às configurações de rede, precisa-se escolher se a *Lambda* pertencerá a alguma rede privada ou não. Mais especificamente nesse projeto, caso a *Lambda* precisasse acessar o banco de dados (o qual se encontra em uma rede privada), ela deve estar dentro dessa rede privada. Dessa forma, precisou-se dividir as operações de consulta ao banco de dados e *download* dos arquivos no *bucket*, visto que o acesso ao S3 necessita ser de fora de qualquer rede privada.

Além dessas configurações, é permitido definir qual o método que será invocado ao disparar a *Lambda*. Por padrão e para as funções desenvolvidas, o método executado é o "*lambda_handler*" do arquivo "*lambda_function.py*".

6.4.1 Coletar números de série

A função *Lambda* em questão tem como responsabilidade buscar, a partir da lista de identificadores dos grupos recebida como parâmetro, os números de séries cadastrados neles. O código pode ser visto na Figura 25.

A *Lambda* inicia coletando do objeto "*event*" a lista de grupos (dada pelo parâmetro "*groupIds*") nos parâmetros passados pela URL através da chave "*queryStringParameters*". Em seguida, um objeto da classe "*TixInspectorGetSerialNumbers*" é criado passando a lista de grupos como argumento. O construtor dessa classe guarda o atributo "*client*", o qual define o recurso da AWS será utilizado; a tabela do *DynamoDB* que será consultada; e a lista de identificadores dos grupos.

Então, para cada grupo, executa-se o método "*get_serial_numbers*", o qual executa a função "*get_item*" no objeto da tabela de grupos, que, por sua vez, recebe a chave do grupo de interesse e retorna um objeto contendo o item (linha) da tabela. Por fim, o método coleta os números de série de dentro do objeto guardado na coluna "*policies*" e retorna uma lista com esses números de série.

Ao fim do laço de repetição, tem-se uma lista com todos os números de série de todos os grupos, a qual é adicionada ao corpo da resposta da função *Lambda*. Com a montagem do JSON de resposta completa, a *Lambda* retorna o objeto "*response_object*" para a API.

Figura 25 – Função *Lambda* para coletar os números de série cadastrados para uma lista de grupos.

```python
class TixInspectorGetSerialNumbers:
    def __init__(self, group_ids):
        self.client = boto3.resource('dynamodb')
        self.table = self.client.Table('Groups')
        self.group_ids = group_ids

    def get_serial_numbers(self, group_id):
        try:
            item = self.table.get_item(
                Key={
                    'id': group_id
                })['Item']
            seriais = item['policies'][0]['resources'][0]['resourceValues']
            return list(seriais)
        except KeyError:
            return []

def lambda_handler(event, context):
    group_ids_raw = event['queryStringParameters']['groupIds']
    group_ids = json.loads(unquote(group_ids_raw))

    tix_inspector_get_serial_numbers = TixInspectorGetSerialNumbers(group_ids)
    serial_numbers = []
    for id in tix_inspector_get_serial_numbers.group_ids:
        serial_numbers.extend(
            tix_inspector_get_serial_numbers.get_serial_numbers(id))

    response_object = {}
    response_object['statusCode'] = 200
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = json.dumps(str(serial_numbers))

    return response_object

```

Fonte: Arquivo pessoal.

6.4.2 Publicar comando

Essa *Lambda* tem a responsabilidade de enviar uma mensagem/comando para o tópico de comandos do *IoT* do computador de bordo escolhido, cujo número de série é passado como parâmetro do método da API. O código dessa função *Lambda* está presente na Figura 26.

Quando ocorre o disparo, resgata-se o número de série na chave "*queryString-Parameters*", o qual remete aos parâmetros passados na URL do método da API e pertence ao JSON do evento ("*event*"). Em seguida, prepara-se o corpo da resposta do método da API e cria-se um novo objeto da classe "*TixInspectorFileRequester*", cujo construtor guarda o número de série e o tópico, e define o cliente dos serviços AWS, ou seja, o serviço de interesse ("*iot-data*" se refere ao *IoT Core*) e a região em que está implantado.

Então, invoca-se o método referente à publicação da mensagem, que, por sua vez, chama o método que cria o *payload* da mensagem *IoT*. Utiliza-se uma mensagem modelo, a qual é um JSON que contém apenas as chaves e valores nulos. Assim, o método abre e carrega a mensagem modelo para a variável "*datastore*", cujos valores são preenchidos, em seguida, de acordo com o valor das variáveis de ambiente (linhas 1 a 5 do código). Os valores das variáveis "*DBUS_OBJECT*", "*DBUS_INTERFACE*" e "*DBUS_METHOD*" serão explicados no capítulo 7, pois foram definidas durante a implementação do subsistema embarcado.

Ao fim da criação do *payload*, invoca-se o método "*publish*" do cliente para publicar o comando no tópico definido. Por fim, o objeto de resposta do método da API é criado e retornado.

6.4.3 Manipulador de novo arquivo

A *Lambda* em questão possui como evento de disparo a inserção de um novo arquivo, cuja chave do S3 possui sufixo "*inspection.json*", no *bucket* onde são armazenados os pacotes enviados pelo *Cloud Sync*. Então, deve inserir um registro no banco de dados com as informações acerca desse novo arquivo. O código dessa *Lambda* está apresentado na Figura 27.

Quando a função é disparada, guarda-se o *timestamp* atual, além de buscar no JSON do evento o nome do *bucket* e a chave do arquivo inserido. Então cria-se uma nova instância da classe "*TixInspectorNewPkgHandler*", cujo construtor armazena o *bucket*, chave e número de série relativo ao novo arquivo. Além disso, cria-se a conexão com o banco de dados através do método "*db_connect*", o qual utiliza a biblioteca "*psycopg2*" para acesso aos bancos de dados *PostgreSQL* em *Python*. Os parâmetros para conexão com o banco de dados são definidos via variáveis de ambiente, as quais são resgatadas nas linhas 1 a 5 do código.

Então, o método "*insert_pkg_in_db*" é invocado a fim de adicionar um novo registro à tabela do banco de dados. Primeiro, identifica-se o número identificador máximo presente na tabela, e, portanto, executa-se a *query* para inserir um novo registro utilizando os dados armazenados na instância da classe e o identificador seguinte.

6.4.4 *Scanear* novos arquivos

Essa função *Lambda* tem o propósito de retornar qual(is) é(são) o(s) arquivo(s) mais recente(s) para dado computador de bordo, de acordo com a necessidade do usuário. Com ela é possível verificar quais são os arquivos mais novos que dado horário, mas também permite resgatar apenas o arquivo mais recente, cenário utilizado para saber qual o último momento em que um arquivo de inspeção foi recebido na nuvem de dado computador de bordo.

Para isso, necessita-se do número de série e, no primeiro caso, do *timestamp* do arquivo recebido pelo usuário no aplicativo, os quais servirão como filtro dos registros no banco de dados. O código dessa função *Lambda* é exposto na Figura 28.

Como na *Lambda* explicada na seção 6.4.2, adquire-se os parâmetros da chamada à API pelo JSON do evento que disparou a *Lambda*. Caso não tenha sido passado o parâmetro de *lastTimestamp*, significa que a *Lambda* foi chamada para o segundo caso, portanto o número máximo necessário de registros da tabela é de apenas um e o *timestamp* mínimo é zero. Caso contrário, o limite possui valor cinco, a fim de evitar retornar uma grande quantidade de dados antigos sem necessidade.

Em seguida, cria-se um objeto da classe "*TixInspectorScanNewFiles*", cujo construtor guarda as informações da chamada e cria a conexão com o banco de dados da mesma forma que na Lambda da seção 6.4.3.

Então, o método "*scan_new_files*" é invocado, o qual executa uma *query* que seleciona os dados que possuem *timestamp* maior que o último *timestamp* recebido pelo usuário, e que pertencem ao computador de bordo escolhido, ou seja, possui o mesmo número de série. Para cada registro, adiciona-se um dicionário contendo os seus dados à lista de novos arquivos. Por fim, são criados o corpo da resposta e a resposta em si, retornando-a à API.

6.4.5 Baixar novos arquivos

Nessa função, deseja-se realizar o *download* dos arquivos passados como parâmetro e retorná-los ao aplicativo *mobile*, o qual irá interpretar a resposta e apresentar os dados ao usuário. A Figura 29 apresenta o código fonte da função em questão.

Primeiro, resgata-se do evento o JSON retornado pela *Lambda* apresentada na seção 6.4.4, passando-o ao construtor da classe "*TixInspectorFetchNewFiles*", o qual guarda esse JSON e cria o cliente para o serviço S3.

Para cada arquivo que se deseja realizar o download, executa-se o método "*get_s3_pkg*" que tenta buscar o arquivo no *bucket* através da chave especificada, retornando um dicionário com o nome do arquivo e o seu conteúdo. Caso, por algum motivo, o arquivo não exista mais no *bucket*, será levantada uma exceção do tipo "*ClientError*", fazendo com que o método retorne um dicionário vazio.

Caso o nome do arquivo seja "*inspection.json*", o seu conteúdo será armazenado no dicionário de retorno ("new_files") como valor da chave dada pelo nome do arquivo e, dentro dessa, como valor da chave dada pelo *timestamp* do arquivo. Caso contrário, o conteúdo será armazenado como valor da chave dada pelo nome do arquivo, caso o conteúdo seja mais recente. Essa distinção é feita, pois, para arquivos de inspeção, deseja-se que todos os arquivos sejam retornados ao aplicativo *mobile* a fim de serem agrupados em um só, devido ao fato de ser possível possuírem conteúdos diferentes.

Figura 26 – Função *Lambda* para requisitar um novo arquivo de inspeção.

```python
MESSAGE_FILE = os.getenv('message_file')
DBUS_OBJECT = os.getenv('dbus_object')
DBUS_INTERFACE = os.getenv('dbus_interface')
DBUS_METHOD = os.getenv('dbus_method')
REPLY_TOPIC = os.getenv('reply_topic')

class TixInspectorFileRequester:
    def __init__(self, serial):
        self.client = boto3.client('iot-data', region_name='region')
        self.serial = serial
        self.topic = 'commands_topic_{}'.format(serial)

    def create_payload_message(self):
        with open(MESSAGE_FILE, "r") as pattern_message:
            datastore = json.load(pattern_message)
            payload_id = int(time.time())

            datastore["id"] = payload_id
            datastore["object"] = DBUS_OBJECT
            datastore["interface"] = DBUS_INTERFACE
            datastore["member"] = DBUS_METHOD
            datastore["reply_topic"] = REPLY_TOPIC
            datastore["params"].append(param)

            payload = json.dumps(datastore)
            return payload.encode('utf-8')

    def publish_iot_message(self):
        payload = self.create_payload_message()
        print(payload)
        response = self.client.publish(
            topic=self.topic,
            qos=0,
            payload=payload
        )
        return response


def lambda_handler(event, context):
    serial_number = event['queryStringParameters']['serialNumber']

    print("Serial number=" + serial_number)

    request_response = {}
    request_response['serialNumber'] = serial_number
    request_response['message'] = "test message"

    tix_inspector_file_requester = TixInspectorFileRequester(serial_number)
    print(tix_inspector_file_requester.publish_iot_message())

    response_object = {}
    response_object['statusCode'] = 200
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = json.dumps(request_response)

    return response_object

```

Fonte: Arquivo pessoal.

Figura 27 – Função *Lambda* para manipular um novo arquivo de inspeção.

```python
DB_HOST = os.getenv('db_host')
DB_USER = os.getenv('db_user')
DB_PASSWORD = os.getenv('db_password')
DB_PORT = os.getenv('db_port')
DB_NAME = os.getenv('db_name')

class TixInspectorNewPkgHandler:
    def __init__(self, bucket, key, timestamp):
        self.bucket = bucket
        self.key = key
        self.serial = key.split('/')[0].split('_extracted')[0]
        self.timestamp = timestamp
        self.db_connection = self.db_connect(
            DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME)
        self.db_cursor = self.db_connection.cursor()

    def db_connect(self, user, password, host, port, db_name):
        try:
            conn = psycopg2.connect(host=host, port=port,
                                    user=user, password=password,
                                    database=db_name, connect_timeout=5)
        except psycopg2.Error as e:
            print("ERROR: Unexpected error: Could not connect to Postgresql instance.")
            print(e)
            sys.exit()
        else:
            return conn

    def insert_pkg_in_db(self):
        max_id_sql = """
            SELECT max(id) FROM inspection_file
        """
        self.db_cursor.execute(max_id_sql)
        max_id = self.db_cursor.fetchone()[0]
        max_id = max_id if max_id is not None else 0

        self.db_cursor.execute("""
            INSERT INTO inspection_file (
                id, serial_number, bucket, file_key, file_timestamp
            ) VALUES (
                %s, %s, %s, %s, %s
            )
        """, (max_id + 1, self.serial, self.bucket, self.key, self.timestamp))
        self.db_connection.commit()

def lambda_handler(event, context):
    timestamp = int(time.time())
    bucket_name = event['Records'][0]['s3']['bucket']['name']
    key = event['Records'][0]['s3']['object']['key']

    tix_inspector_new_pkg_handler = TixInspectorNewPkgHandler(
        bucket_name, key, timestamp)
    print("New package uploaded at {} to bucket '{}' with key '{}' from serial number '{}'".format(
        tix_inspector_new_pkg_handler.timestamp,
        tix_inspector_new_pkg_handler.bucket,
        tix_inspector_new_pkg_handler.key,
        tix_inspector_new_pkg_handler.serial
    ))
    tix_inspector_new_pkg_handler.insert_pkg_in_db()
    print('Package registered successfully in the database')

```

Fonte: Arquivo pessoal.

Figura 28 – Função *Lambda* para *scanear* novos arquivos no banco de dados.

```python
# Mesmas variáveis de ambiente da Lambda tix_inspector_new_pkg_handler
class TixInspectorScanNewFiles:
    def __init__(self, serial_number, last_timestamp, limit):
        self.serial = serial_number
        self.last_timestamp = last_timestamp
        self.limit = limit
        self.db_connection = self.db_connect(
            DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME)
        self.db_cursor = self.db_connection.cursor()

    def db_connect(self, user, password, host, port, db_name):
        # O mesmo método da classe TixInspectorNewPkgHandler

    def scan_new_files(self):
        self.db_cursor.execute("""
            SELECT id, bucket, file_key, file_timestamp
            FROM inspection_file
            WHERE file_timestamp > %s AND serial_number = %s
            ORDER BY file_timestamp DESC
            LIMIT %s
        """, (self.last_timestamp, self.serial, self.limit))
        response = self.db_cursor.fetchall()
        new_file_list = []
        for id_, bucket, key, timestamp in response:
            new_file_list.append({
                "id": id_,
                "bucket": bucket,
                "key": key,
                "timestamp": timestamp
            })
        return new_file_list

def lambda_handler(event, context):
    serial_number = event['queryStringParameters']['serialNumber']
    try:
        last_timestamp = event['queryStringParameters']['lastTimestamp']
        limit = 5
    except KeyError:
        last_timestamp = 0
        limit = 1

    tix_inspector_fetch_new_files = TiXInspectorScanNewFiles(
        serial_number, last_timestamp, limit)
    new_pkg_list = tix_inspector_fetch_new_files.scan_new_files()

    request_response = {}
    request_response['serialNumber'] = serial_number
    request_response['oldLastTimestamp'] = last_timestamp
    request_response['newPkgs'] = new_pkg_list

    response_object = {}
    response_object['statusCode'] = 200
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = json.dumps(request_response)

    return response_object

```

Fonte: Arquivo pessoal.

Figura 29 – Função *Lambda* para baixar novos arquivos.

```python
class TixInspectorFetchNewFiles:
    def __init__(self, pkg_info):
        self.pkg_info = pkg_info
        self.s3_client = boto3.client('s3', region_name='region')

    def get_s3_pkg(self, bucket, key):
        try:
            input_file = self.s3_client.get_object(Bucket=bucket, Key=key)
            input_content = json.loads(
                input_file['Body'].read().decode('utf-8'))
            return {
                "name": key.split('/')[-1],
                "content": input_content
            }
        except ClientError:
            return {}

def lambda_handler(event, context):
    pkg_info_raw = event['queryStringParameters']['newPkgs']
    pkg_info = json.loads(unquote(pkg_info_raw))

    tix_inspector_fetch_new_files = TixInspectorFetchNewFiles(
        pkg_info)

    new_files = {}
    for pkg_file in tix_inspector_fetch_new_files.pkg_info:
        package = tix_inspector_fetch_new_files.get_s3_pkg(
            pkg_file['bucket'], pkg_file['key'])
        if len(package) == 0:
            continue

        file_name = package['name']
        if file_name == "inspection.json":
            if file_name not in new_files.keys():
                new_files[file_name] = {}
            new_files[file_name][pkg_file['timestamp']] = package['content']
        else:
            if file_name in new_files.keys():
                if int(new_files[file_name]['timestamp']) >= int(pkg_file['timestamp']):
                    continue
            new_files[file_name] = {
                "content": package['content'],
                "timestamp": pkg_file['timestamp']
            }

    response_object = {}
    response_object['statusCode'] = 200
    response_object['headers'] = {}
    response_object['headers']['Content-Type'] = 'application/json'
    response_object['body'] = json.dumps(new_files)

    return response_object
```

Fonte: Arquivo pessoal.

7 SISTEMA EMBARCADO

Para o projeto do *TiX Inspector*, o computador de bordo é o objeto de interesse, visto que é o sistema sobre o qual é feita a inspeção por algum motivo. Quanto ao subsistema do projeto contido dentro do computador de bordo, este deve ser acionado por uma mensagem IoT no tópico de comandos, resgatar as informações das fontes corretas, tratar os dados, organizar em um arquivo de inspeção e enviá-lo para a nuvem. A fim de ilustrar a sequência dessas ações considerando a estrutura interna do software do computador de bordo e sua interação com o subsistema em nuvem, criou-se a Figura 30, a qual será explicada com mais detalhe no decorrer deste capítulo.

Neste subsistema, despendeu-se mais tempo e esforço para identificar a fonte dos dados de interesse, resgatá-los e tratá-los (comentado detalhadamente nas seções 7.1 e 7.2). Além disso, precisou-se interagir com os processos já implementados no computador de bordo para construir o fluxo de funcionamento. Ademais, houve a necessidade de configurar corretamente o computador de bordo para permitir o uso do *TiX Inspector*, o que será explicado na seção 7.3.

Vale ressaltar que, nesse subsistema, foi desenvolvido pelo autor apenas as entidades coloridas na Figura 30, as quais interagem com entidades já presentes no *software* do computador de bordo.

7.1 FONTES DOS DADOS

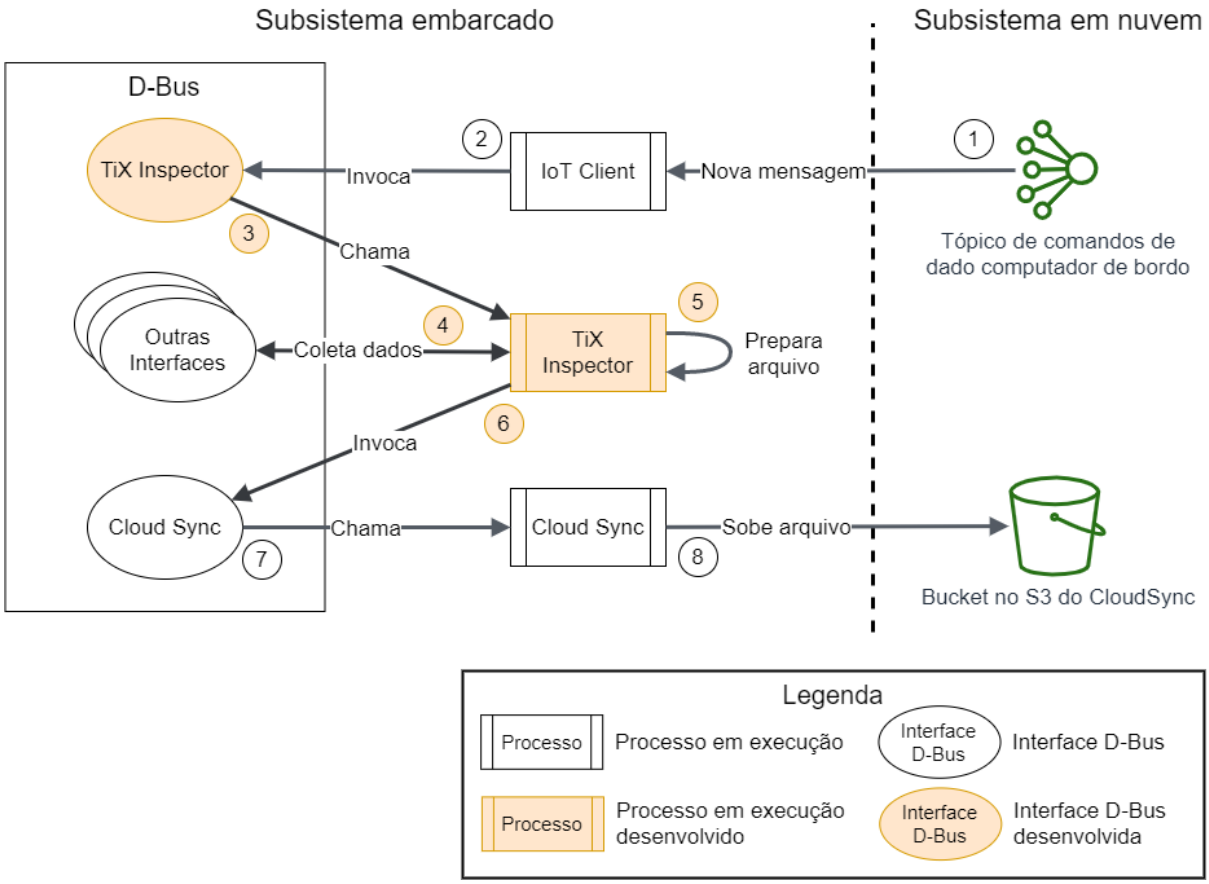
A partir do que foi definido no início do projeto, comentado na seção 4.3, estudou-se o código e os arquivos contidos no *software* embarcado, além do conteúdo do arquivo de diagnóstico, a fim de identificar a fonte de cada uma das informações escolhidas.

Concluiu-se que os dados a serem resgatados são advindos de três fontes principais: arquivos, resultados de comandos *Linux* e do barramento *D-Bus*. Além disso, definiu-se qual será o tipo de *widget* em que cada dado será apresentado. Com as fontes para cada dado e o seu respectivo tipo de *widget* definidos, construiu-se a tabela mostrada na Tabela 2, com o intuito de visualizar melhor as informações encontradas.

Manteve-se a organização dos dados definida anteriormente na seção 4.3, os quais são determinados nas colunas “Módulo” e “Seção”. Dentro de cada seção, contém-se informações, dadas pela coluna “Informação”, as quais foram agrupadas quando estas possuem a mesma fonte.

Na coluna “Fonte” são apresentadas as fontes relativas às informações apresentadas na coluna à esquerda. Por fim, na coluna “*Widget*”, definiu-se o tipo de *widget* que irá apresentar tal(is) dado(s).

Figura 30 – Diagrama do fluxo de funcionamento do subsistema embarcado.



Fonte: Arquivo pessoal.

7.2 IMPLEMENTAÇÃO DO SERVIÇO

Para implementar o serviço do *TiX Inspector*, utilizou-se a linguagem de programação *Python*, usando como base um serviço já implementado no *software*, o qual também está implementado em *Python* e possui uma interface *D-Bus* relacionada.

O serviço do *TiX Inspector* é definido a partir de uma classe, cujo construtor inicializa um objeto *D-Bus* com base no nome da interface e do objeto passados como parâmetro. Além disso, a classe define quais são os métodos da interface, assim como o seu comportamento, o que, para o serviço do *TiX Inspector*, foi criado apenas o método "*generate_inspection_file*". Esse método repassa a invocação para o serviço do *TiX Inspector* (ação número 3 da Figura 30), o qual continua o fluxo do subsistema.

O método da D-Bus do *TiX Inspector* é invocado a partir do serviço *IoT Client* (ação número 2), que, por sua vez, é disparado por uma nova mensagem no tópico de comandos do IoT do computador de bordo em questão (ação número 1). Essa mensagem contém informações acerca de qual a interface, objeto e método deve ser invocado, assim como os seus parâmetros e qual o tópico de resposta do comando.

Apesar da figura apenas mostrar, na ação número 4, a busca dos dados no barramento D-Bus, o serviço do *TiX Inspector* realiza as ações necessárias para resgatar os dados das três fontes definidas anteriormente. Quanto à implementação dessa parte, decidiu-se dividir, em geral, o código fonte seguindo a organização de módulo e seção, tal que os módulos possuem um módulo Python (arquivo *.py*) e as seções possuem uma função dentro do módulo ao qual pertence. Cada função possui a responsabilidade de resgatar todas as informações relativas a sua seção, organizar no formato padrão para ser lido posteriormente e retornar um dicionário com os devidos dados tratados. Dessa forma, o código fica organizado seguindo uma estrutura de composição de funções, similar ao utilizado no subsistema *mobile*, podendo ser expandido facilmente no futuro com a adição de novas funções e módulos.

Por fim, munido do dicionário completo com os dados coletados, o serviço do *TiX Inspector* cria um arquivo de inspeção no armazenamento do computador de bordo e invoca o método de envio de arquivos para nuvem da *D-Bus* do serviço *Cloud Sync*, o qual chama um método do serviço que adiciona o arquivo recebido como parâmetro à lista de arquivos pendentes do serviço. Então, o *Cloud Sync* sobe para a nuvem o arquivo de inspeção.

7.3 CONFIGURAÇÃO DO TITANIUM

A fim de instalar o subsistema embarcado do *TiX Inspector* no computador de bordo, decidiu-se que é necessário a instalação de um CTI (arquivo de configuração) customizado para tal. Apesar de o conteúdo da atualização ser um fragmento de *software*, acrescentar o *TiX Inspector* no pacote de atualização de *software* (UTI) permitiria somente a inspeção dos computadores de bordo com a versão mais atual e as versões subsequentes, justificando a escolha pelo CTI.

Devido a característica do serviço *IoT Client* de apenas permitir o recebimento de mensagens de comando relacionadas à *D-Bus* (seja chamada de método ou inscrição em sinal), precisou-se criar um barramento e uma interface para o serviço do *TiX Inspector*. Para tal, foi necessário incluir um arquivo de configuração do serviço do sistema e do serviço da *D-Bus* ao CTI e instalá-los corretamente.

Como o sistema está em fase de validação do conceito, implementou-se o subsistema em ambiente de desenvolvimento (*dev*). Portanto, é necessário a inserção dos certificados de *dev* no computador de bordo, com o intuito deste se autenticar com todos os sistemas em nuvem da AWS.

Além disso, algumas configurações gerais do sistema devem ser alteradas para permitir o uso do *TiX Inspector*. Quanto ao serviço *IoT Client*, precisa-se definir que serão utilizadas configurações do ambiente de *dev* e acrescentar a interface *D-Bus* do *TiX Inspector* à lista de interfaces *D-Bus* cujo acesso pela nuvem é permitido, visto que apenas algumas são permitidas por padrão. Quanto ao *Cloud Sync*, precisa-se

alterar o *bucket* de envio dos pacotes para o *bucket* do ambiente de desenvolvimento.

Preparou-se o *script* de instalação, o qual realiza as seguintes operações:

1. Cria um backup das configurações atuais.
2. Copia o arquivo de configuração do serviço do sistema e cria um *link* simbólico para ele.
3. Adiciona o comando de inicializar o serviço do *TiX Inspector* na rotina de inicialização do sistema.
4. Adiciona o arquivo de configuração do serviço da D-Bus.
5. Copia os módulos *.py*.
6. Insere os certificados de *dev*.
7. Altera as configurações do *IoT Client* e *Cloud Sync*.

O *script* de desinstalação reverte cada uma das operações definidas na instalação, restaurando a configuração anterior.

Tabela 2 – Fontes dos dados.

Módulo	Seção	Informação	Fonte	Widget
General	Basic Info	Baseboard ID Board model Is blackbox Mobile modem model type OEM Password Serial TiX	Comando	Lista
General	Basic Info	Timezone	D-Bus	Lista
General	Storage	Total Sporadic Periodic Repository	Arquivo	Lista
General	Coredumps	Context Timestamp Size	Comando	Tabela
General	Activations	-	Comando	Lista
General	Installed versions	Last configuration (.cti) Current configuration (.cti) Last software (.uti) Current software (.uti)	D-Bus	Lista
General	Running processes	-	Arquivo	Tabela
Connectivity	General	Saved connections Devices	Arquivo	Tabela
Connectivity	Wi-Fi	Devices	Arquivo	Tabela
Connectivity	Wi-Fi	Information Signal	D-Bus	Lista
Connectivity	Mobile	Information Configuration Signal	D-Bus	Lista
Connectivity	GNSS	Antenna Offset Configuration GNSS Info GNSS Terrastar Info Nudge Position	D-Bus	Lista
Connectivity	Cloud	Connected	D-Bus	Lista
Connectivity	Cloud	Pending files	D-Bus	Lista
Connectivity	Cloud	Certificates	Comando	Lista
Monitoring	Configuration	PPR count Snapshot ID Service order Cost center Role Vehicle type Vehicle Implement type Implement Implement width	Banco de dados	Lista
Monitoring	Sensors	-	Banco de dados + D-Bus	Tabela

Fonte – Arquivo pessoal.

8 RESULTADOS

No capítulo em questão, apresenta-se uma análise dos resultados obtidos no projeto desenvolvido. Primeiramente, faz-se a verificação quanto ao cumprimento dos requisitos do sistema definidos na seção 4.2. Em seguida, expõem-se os testes realizados e as conclusões obtidas através deles. Por fim, avalia-se a solução em um contexto mais geral, tendo em vista as perspectivas de uso da aplicação dentro dos processos da empresa.

8.1 CUMPRIMENTO DOS REQUISITOS

Uma forma de julgar os resultados do projeto é revisitando os requisitos do sistema definidos na sua fase inicial e verificando se eles foram de fato contemplados na solução final.

- Quanto ao uso de recursos gráficos, fez-se uso, mesmo que de forma simplificada, dos *widgets* em dois formatos (listas e tabelas) para mostrar os dados de saúde do computador de bordo ao usuário.
- A atualização dos dados é disparada de acordo com o comando do usuário e seus valores são modificados com a finalização do download e o tratamento dos arquivos de inspeção. Assim, a interface muda à medida que dados mais atuais são recebidos.
- Usou-se uma biblioteca de armazenamento de dados em memória não volátil, como comentado na seção 5.6, para permitir a troca do computador de bordo inspecionado, resgatando os últimos dados. Além disso, utilizou-se essa biblioteca para armazenar localmente outras informações importantes entre as execuções do aplicativo.
- Adicionou-se um botão em cada seção, o qual permite ao usuário favoritar as seções que deseja visualizar mais facilmente no futuro.
- Preparou-se toda a infraestrutura em nuvem para que fosse possível requisitar um novo arquivo de inspeção, guardar arquivos na nuvem, baixar os arquivos novos, além de verificar a data do último arquivo enviado pelo computador de bordo.
- Como explicado na seção 5.8, integrou-se com os cadastros de usuário já empregados em outros sistemas da Hexagon, a fim de reutilizar o *login* e evitar a criação de mais contas desnecessárias.

- Através da transformação dos dados para o formato HTML, foi possibilitada a criação de um arquivo PDF contendo os dados de inspeção atuais, os quais foram chamados de *snapshots* e explicados na seção 5.7.
- Como apresentado no capítulo 7, desenvolveu-se um serviço em *Python* para o *TiX Inspector* a fim de resgatar todos os dados definidos na seção 4.3.
- Ao utilizar o *framework React Native*, garantiu-se que o aplicativo *mobile* fosse utilizado nos dois principais sistemas operacionais para *smartphones* da atualidade.
- Expôs-se, no capítulo 6, a infraestrutura em nuvem implementada utilizando a plataforma AWS, cumprindo o requisito não funcional definido anteriormente.
- Definiu-se uma estrutura de composição para organização dos dados do computador de bordo, a qual consiste em módulos, seções e *widgets*. Dessa forma, dividiram-se os dados em contextos, auxiliando na usabilidade do usuário.
- Além disso, essa estrutura foi implementada de forma que a alteração do sistema para adição, mudança e remoção de dados, assim como da sua organização, possa ser feita rapidamente e sem prejudicar as outras partes do sistema.
- Por fim, desde a construção dos protótipos das telas do aplicativo móvel, tomou-se como base as interfaces empregadas nos aplicativos mais utilizados atualmente. Assim, acredita-se que a interface ficou relativamente intuitiva para os usuários.

8.2 TESTES REALIZADOS

Durante a implementação do projeto, realizaram-se testes de cada subsistema de forma isolada para cada funcionalidade que ia sendo adicionada ao projeto. Somente quando os subsistemas separados tinham uma implementação sólida de determinada função no sistema é que se prosseguiu para um teste em conjunto, priorizando, primeiramente, os testes em pares de subsistemas antes do teste completo. Assim, ocorreram menos falhas durante os testes utilizando todos os subsistemas, pois cada uma das partes havia sido devidamente testada anteriormente.

Alguns cenários de testes internos foram utilizados durante o desenvolvimento do projeto, variando o computador de bordo; a versão de software instalada nele, sendo uma próxima da mais recente e outra lançada há quase um ano; o sistema operacional do smartphone, tanto *iOS* quanto *Android*; e modelos de celulares diferentes, sendo uns mais antigos com o dispositivo físico, mas também modelos mais novos através de simulador.

Para o cenário em que o aplicativo estava sendo executado no sistema *iOS*, ocorreram problemas na apresentação da interface, o que afetou relativamente a usabilidade do aplicativo. Por outro lado, as funcionalidades se comportaram como o esperado. Com isso, por falta de equipamento e tempo disponível para testes em ambos os sistemas operacionais, não foi possível realizar as correções necessárias para o sistema *iOS*, priorizando-se a implementação para o sistema *Android*.

Em todos os demais cenários o sistema se comportou bem, tendo apenas variações de performance de acordo com o nível de *hardware* disponível no *smartphone*.

Por fim, finalizado o desenvolvimento do sistema, efetuaram-se testes com colaboradores das equipes da cadeia de suporte e da garantia de qualidade, os quais são os possíveis usuários do *TiX Inspector* em cenários mais próximos do real. Do ponto de vista do sistema em si, os resultados foram considerados satisfatórios, atendendo o seu propósito de funcionamento e cumprindo com todos os casos de uso propostos. Somente foram identificados alguns problemas pontuais com relação à usabilidade, em que a interface não estava suficientemente intuitiva, e pequenos *bugs* - todos com formas de contorno, não prejudicando o fluxo de uso completamente. Estima-se que todas as propostas de mudanças sugeridas pelos testadores poderão ser implementadas sem grande esforço e/ou retrabalho no futuro. Também foram sugeridos dados de interesse que poderiam estar presentes e serem úteis para o dia-a-dia, facilitando algumas tarefas comuns.

8.3 AVALIAÇÃO DA SOLUÇÃO

Quanto ao que havia sido proposto no início do projeto, acredita-se que o projeto atingiu os requisitos e cumpriu com os casos de uso propostos. Julga-se que o sistema se comportou bem durante os testes em diversos cenários, validando a estrutura de *software* e a utilidade da ferramenta.

Em conversa com os gestores, concluiu-se que a solução possui grande potencial de ser utilizada pelos colaboradores e é uma ótima forma de acelerar o processo de suporte, reduzindo o tempo para resolução de problemas em campo. Além disso, acredita-se que o aplicativo móvel poderá diminuir a atual falta de informações para diagnosticar a causa de algumas falhas nas soluções da Hexagon, com a vantagem de funcionar remotamente.

Viu-se a oportunidade de utilizar os dados coletados no computador de bordo no novo sistema de acesso remoto que está em desenvolvimento, como comentado na seção 2.2. Apesar de o contexto de uso ser diferente, acredita-se que pode ser interessante para os usuários do acesso remoto ter algumas informações básicas internas ao computador de bordo, como qualidade da conexão, por exemplo. Dessa forma, esses dois sistemas utilizariam da mesma infraestrutura em nuvem, porém empregando-a em cenários distintos.

9 CONCLUSÃO

Dados os resultados apresentados no capítulo anterior, julga-se que o projeto descrito neste documento teve sucesso, alcançando seus objetivos e validando a ideia inicial. Além disso, sua implementação permite que o uso seja expandido para outras finalidades, sendo considerada uma ferramenta de grande potencial nos processos internos da empresa.

Quanto à relação deste projeto com o curso de graduação de Engenharia de Controle e Automação, acredita-se que as disciplinas de programação e modelagem de sistemas computacionais foram de suma importância para permitir o desenvolvimento do projeto. Além delas, a disciplinas optativas de “Fundamentos de Sistemas de Bancos de Dados” e “Bancos de Dados III” permitiram que fossem manipulados bancos de dados relacionais e não relacionais de forma mais espontânea. Ademais, a optativa de “Integração de Sistemas Corporativos” foi útil para entendimento dos sistemas de computação em nuvem, assim como por permitir uma visão mais macro do sistema e as ligações entre seus subsistemas.

No entanto, pensa-se que a disponibilização de disciplinas optativas mais focadas em desenvolvimento de sistemas *mobile* pudessem ter facilitado a modelagem e a implementação do subsistema móvel, visto que não se possuía nenhum conhecimento prévio sobre o assunto.

Anteriormente a este trabalho, desenvolveu-se um projeto para a mesma empresa durante a disciplina de “Estágio Obrigatório”. Portanto, já havia experiência acerca dos processos e sistemas da Hexagon, o que facilitou as fases iniciais do projeto e a comunicação com os colaboradores.

Como desafio, pode-se citar que o Projeto de Fim de Curso foi inteiramente produzido de forma remota, tendo em vista as medidas restritivas adotadas pela empresa devido à pandemia de Covid-19. Apesar disso, as reuniões e a comunicação com toda a equipe não foram afetadas, mantendo-se o contato durante todo o semestre, assim como o desempenho das atividades normalmente.

Acredita-se que, com a conclusão desse projeto, diversos conhecimentos foram adquiridos no contexto de cada um dos subsistemas com que se teve contato: aplicação embarcada em sistema operacional *Linux*, plataforma de computação em nuvem AWS e aplicativos móveis utilizando o *framework React Native*. Com isso, foi possível agregar experiências importantes para uma formação mais completa.

9.1 TRABALHOS FUTUROS

Dados os *feedbacks* recebidos dos usuários que testaram o sistema ao final do projeto e as ideias que surgiram durante a trajetória de desenvolvimento, acredita-se que os seguintes trabalhos sejam relevantes para o futuro do *TiX Inspector*:

- Acrescentar mais dados resgatados dentro do computador de bordo, como, por exemplo, dados de outros produtos do portfólio da empresa, arquivos de configuração etc. Com isso, o sistema irá contemplar um maior número de casos a serem investigados.
- Implementar as sugestões dadas pelos testadores acerca do funcionamento e interface do aplicativo móvel a fim de melhorar a sua usabilidade.
- Implementar a atualização segmentada dos dados de inspeção, separando os pacotes de acordo com o módulo que o usuário deseja inspecionar. Com isso, reduz-se o consumo de dados e custos dos serviços utilizados da plataforma AWS.
- Implementar outros tipos de *widgets* que representem melhor os dados e sejam menos textuais, facilitando a leitura dos dados e, consequentemente, o entendimento do estado do computador de bordo.
- Desenvolver uma tela diretamente no computador de bordo que mostre os dados de forma simplificada, com o intuito de desacoplar as soluções. Desse modo, caso o usuário tenha acesso direto ao computador de bordo, ele pode verificar os dados de saúde do dispositivo sem o uso de um *smartphone*.
- Melhorar o desempenho do aplicativo *mobile* de forma geral, reduzindo os tempos de carregamento. Assim, o sistema irá se comportar bem em dispositivos mais antigos, atingindo um maior número de dispositivos e usuários.
- Incluir algum sistema inteligente que, por meio dos dados de inspeção e conhecimento prévio de causas de problemas já mapeados, possa informar o usuário da causa do problema, o que reduziria ainda mais o tempo de suporte.
- Permitir inspeção do computador de bordo localmente através de alguma tecnologia de pareamento entre o *smartphone* e o computador de bordo, como *Wi-Fi* ou *Bluetooth*. Com isso, a inspeção não dependerá de fatores externos de conectividade, embora não conte com a vantagem de ser remota.

REFERÊNCIAS

AGRICULTURE, Hexagon. **Sobre**. c2021. Disponível em: https://hexagonagriculture.com/pt-br/about. Acesso em: 22 fev. 2021.

ARCHLINUX. **Core dump**. Mar. 2021. Disponível em: https://wiki.archlinux.org/index.php/Core_dump. Acesso em: 14 mar. 2021.

ASIRI, Sidath. **Flux and Redux**. Abril 2018. Disponível em: https://medium.com/@sidathasiri/flux-and-redux-f6c9560997d7. Acesso em: 24 fev. 2021.

B.V., Framer. **The prototyping tool for teams**. c2021. Disponível em: https://www.framer.com/. Acesso em: 10 fev. 2021.

BROWN, DeShawn. **Mobile App Development: Native, Hybrid, and React Native**. Disponível em: https://lithiosapps.com/mobile-app-development-native-hybrid-and-react-native/. Acesso em: 11 fev. 2021.

BUNA, Samer. **The difference between Flux and Redux**. Jan. 2017. Disponível em: https://medium.com/edge-coders/the-difference-between-flux-and-redux-71d31b118c1. Acesso em: 26 fev. 2021.

CANGUÇU, Raphael. **O que são Requisitos Funcionais e Requisitos Não Funcionais?** Fevereiro 2021. Disponível em: https://codificar.com.br/requisitos-funcionais-nao-funcionais/. Acesso em: 19 mar. 2021.

CARBONNELLE, Pierre. **PYPL PopularitY of Programming Language**. c2020. Disponível em: https://pypl.github.io/PYPL.html. Acesso em: 12 fev. 2021.

CHIARELLI, Andrea. **The functional side of React**. Outubro 2018. Disponível em: https://medium.com/@andrea.chiarelli/the-functional-side-of-react-229bdb26d9a6. Acesso em: 22 fev. 2021.

CHINNATHAMBI, Kirupa. **Understanding WebViews**. Disponível em: https://www.kirupa.com/apps/webview.htm. Acesso em: 10 fev. 2021.

COCAGNE, Tom. **DBus Overview**. Mar. 2013. Disponível em: https://pythonhosted.org/txdbus/dbus_overview.html. Acesso em: 16 mar. 2021.

ELLIOTT, Eric. **What is a Pure Function?** Mar. 2016. Disponível em: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-pure-function-d1c076bec976. Acesso em: 22 fev. 2021.

ELLIOTT, Eric. **What is Functional Programming?** Jan. 2017. Disponível em: https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0. Acesso em: 22 fev. 2021.

ESRI. **ESRI Shapefile Technical Description**. [*S.l.*], jul. 1998. Disponível em: https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf. Acesso em: 24 fev. 2021.

FREEDESKTOP.ORG. **What is D-Bus?** Dezembro 2020. Disponível em: https://www.freedesktop.org/wiki/Software/dbus/. Acesso em: 16 mar. 2021.

GOLLWITZER, Zach. **Imperative vs. Declarative Programming (procedural, functional, and OOP)**. Jan. 2020. Disponível em: https://zach-gollwitzer.medium.com/imperative-vs-declarative-programming-procedural-functional-and-oop-b03a53ba745c. Acesso em: 22 fev. 2021.

GRIFFITH, Chris. **What is Hybrid App Development?** c2021. Disponível em: https://ionic.io/resources/articles/what-is-hybrid-app-development. Acesso em: 10 fev. 2021.

GUPTA, Abhay. **Introduction To Redux (using React Native)**. Jul. 2017. Disponível em: https://medium.com/@abhayg772/introduction-to-redux-using-react-native-a8f1e8778333. Acesso em: 26 fev. 2021.

HEXAGON. **Hexagon Overview**. fevereiro 2021. Disponível em: https://bynder.hexagon.com/m/6128ae7405f6a314/original/Hexagon-Corporate-Presentation.pdf. Acesso em: 22 fev. 2021.

KOKJE, Amol. **AWS Lambda Layers**: Why is it so cool? Abril 2019. Disponível em: https://faun.pub/aws-lambda-layers-d07831ff50ea. Acesso em: 10 mar. 2021.

LYNCH, Max. **What are Progressive Web Apps?** Maio 2016. Disponível em: https://blog.ionicframework.com/what-is-a-progressive-web-app/. Acesso em: 10 fev. 2021.

REDUX. **Redux Fundamentals, Part 4: Store**: Middleware. [*S.l.*], c2021. Disponível em: https://redux.js.org/tutorials/fundamentals/part-4-store#middleware. Acesso em: 1 mar. 2021.

SERVICES, Amazon Web. **Amazon API Gateway**. c2021a. Disponível em: https://aws.amazon.com/pt/api-gateway/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Amazon Cognito**. c2021b. Disponível em: https://aws.amazon.com/pt/cognito/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Amazon DynamoDB**. c2021c. Disponível em: https://aws.amazon.com/pt/dynamodb/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Amazon Lambda**. c2021d. Disponível em: https://aws.amazon.com/pt/lambda/?nc2=h_ql_prod_fs_lbd. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **Amazon Relational Database Service (RDS)**. c2021e. Disponível em: https://aws.amazon.com/pt/rds/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do Amazon API Gateway**. c2021f. Disponível em: https://aws.amazon.com/pt/api-gateway/pricing/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do Amazon Cognito**. c2021g. Disponível em: https://aws.amazon.com/pt/cognito/pricing/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do Amazon DynamoDB**. c2021h. Disponível em: https://aws.amazon.com/pt/dynamodb/pricing/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do Amazon RDS**. c2021i. Disponível em: https://aws.amazon.com/pt/rds/pricing/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do AWS IoT Core**. c2021j. Disponível em: https://aws.amazon.com/pt/iot-core/pricing/?nc=sn&loc=4. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **Definição de preço do AWS Lambda**. c2021k. Disponível em: https://aws.amazon.com/pt/lambda/pricing/. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **Perguntas frequentes sobre o Amazon API Gateway**. c2021l. Disponível em: http://aws.amazon.com/pt/api-gateway/faqs/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Perguntas frequentes sobre o Amazon DynamoDB**. c2021m. Disponível em: https://aws.amazon.com/pt/dynamodb/faqs/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Perguntas frequentes sobre o Amazon RDS**. c2021n. Disponível em: https://aws.amazon.com/pt/rds/faqs/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Perguntas frequentes sobre o Amazon S3**. c2021o. Disponível em: https://aws.amazon.com/pt/s3/faqs/?nc=sn&loc=7. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **Perguntas frequentes sobre o AWS Lambda**. c2021p. Disponível em: https://aws.amazon.com/pt/lambda/faqs/. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **Recursos do Amazon Cognito**. c2021q. Disponível em: https://aws.amazon.com/pt/cognito/details/. Acesso em: 12 mar. 2021.

SERVICES, Amazon Web. **Visão geral do AWS IoT Core**. c2021r. Disponível em: https://aws.amazon.com/pt/iot-core/?nc1=h_ls. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **What is Amazon S3?** [*S.l.*], c2021s. Disponível em: https://docs.aws.amazon.com/AmazonS3/latest/dev/Welcome.html. Acesso em: 10 mar. 2021.

SERVICES, Amazon Web. **What is AWS**. c2021t. Disponível em: https://aws.amazon.com/what-is-aws/?nc1=f_cc. Acesso em: 10 mar. 2021.

SOFTWARE, Lucid. **Diagrama de caso de uso UML**: O que é, como fazer e exemplos. c2021. Disponível em: https://www.lucidchart.com/pages/pt/diagrama-de-caso-de-uso-uml. Acesso em: 19 mar. 2021.

SOURCE, Facebook Open. **Componentes e Props**. [*S.l.*], c2021a. Disponível em: https://pt-br.reactjs.org/docs/components-and-props.html. Acesso em: 12 fev. 2021.

SOURCE, Facebook Open. **Core Components and Native Components**. [*S.l.*], c2021b. Disponível em: https://reactnative.dev/docs/intro-react-native-components. Acesso em: 11 fev. 2021.

SOURCE, Facebook Open. **Estado e Ciclo de Vida**. [*S.l.*], c2021c. Disponível em: https://pt-br.reactjs.org/docs/state-and-lifecycle.html. Acesso em: 15 fev. 2021.

SOURCE, Facebook Open. **In-Depth Overview**. [*S.l.*], c2021d. Disponível em: https://facebook.github.io/flux/docs/in-depth-overview/. Acesso em: 24 fev. 2021.

SOURCE, Facebook Open. **In-Depth Overview**: Structure and data flow. [*S.l.*], c2021e. Disponível em: https://facebook.github.io/flux/docs/in-depth-overview/#structure-and-data-flow. Acesso em: 24 fev. 2021.

SOURCE, Facebook Open. **Introdução aos Hooks**. [*S.l.*], c2021f. Disponível em: https://pt-br.reactjs.org/docs/hooks-intro.html. Acesso em: 15 fev. 2021.

SOURCE, Facebook Open. **Introduzindo JSX**. [*S.l.*], c2021g. Disponível em: https://pt-br.reactjs.org/docs/introducing-jsx.html. Acesso em: 12 fev. 2021.

SOURCE, Facebook Open. **React fundamentals**: State. [*S.l.*], c2021h. Disponível em: https://reactnative.dev/docs/intro-react#state. Acesso em: 15 fev. 2021.

SOURCE, Facebook Open. **React life cycle method diagram**. c2021i. Disponível em: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/. Acesso em: 15 fev. 2021.

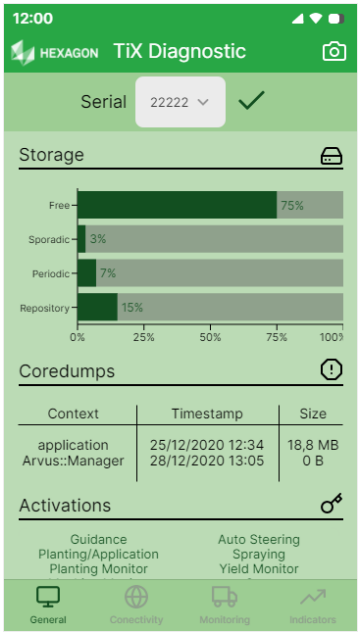
SOURCE, Facebook Open. **React Native**. c2021j. Disponível em: https://reactnative.dev/. Acesso em: 11 fev. 2021.

TECHNOLOGIES, Emorphis. **Progressive Web Apps vs Responsive Web Apps**: How they are Different from Each Other? Nov. 2019. Disponível em: https://medium.com/emorphis-technologies/progressive-web-apps-vs-responsive-web-apps-how-they-are-different-from-each-other-f0cd3640747d. Acesso em: 10 fev. 2021.

APÊNDICE A – TELAS DOS PROTÓTIPOS DO APLICATIVO *MOBILE*

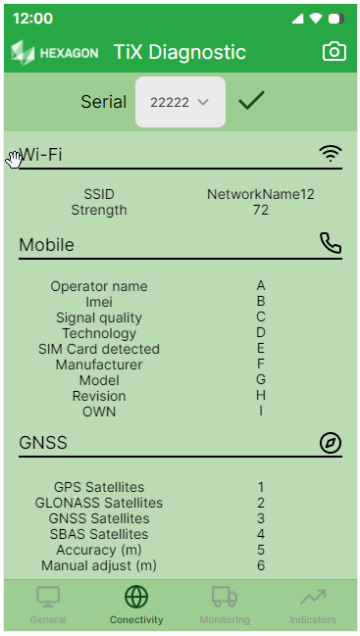
A.1 PROTÓTIPO INICIAL

Figura 31 – Aba "*General*"



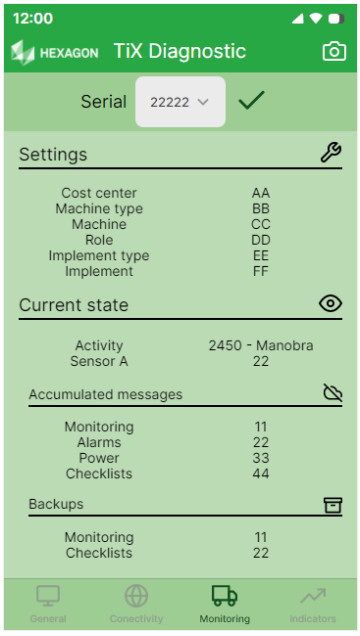
Fonte: Arquivo pessoal.

Figura 32 – Aba "*Connectivity*"



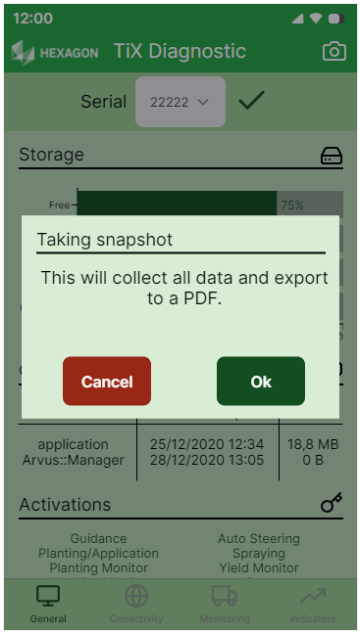
Fonte: Arquivo pessoal.

Figura 33 – Aba "*Monitoring*"



Fonte: Arquivo pessoal.

Figura 34 – Modal de confirmação



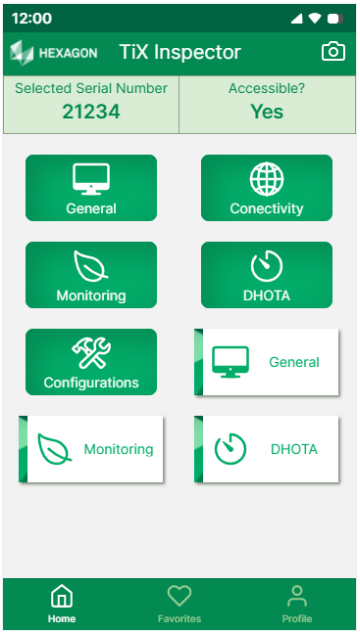
Fonte: Arquivo pessoal.

A.2 PROTÓTIPO FINAL

Figura 35 – Tela de *login*

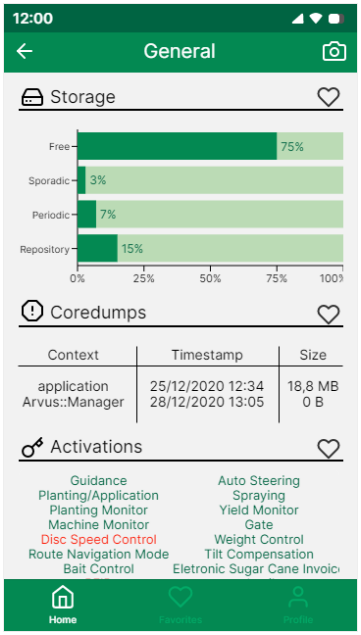
Fonte: Arquivo pessoal.

Figura 36 – Tela inicial



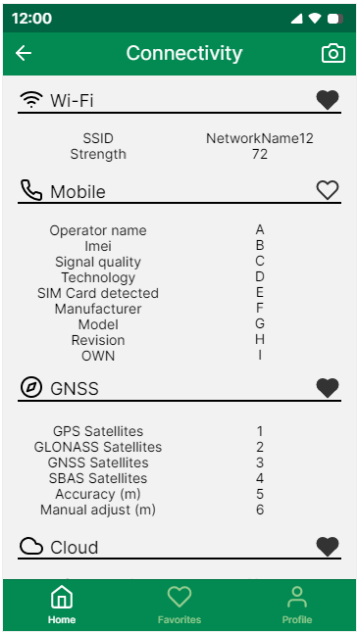
Fonte: Arquivo pessoal.

Figura 37 – Módulo "*General*"



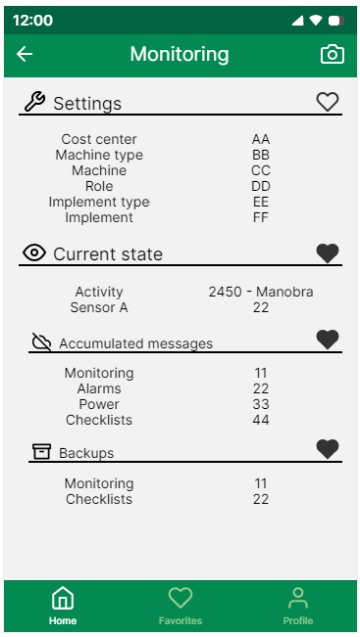
Fonte: Arquivo pessoal.

Figura 38 – Módulo "*Connectivity*"



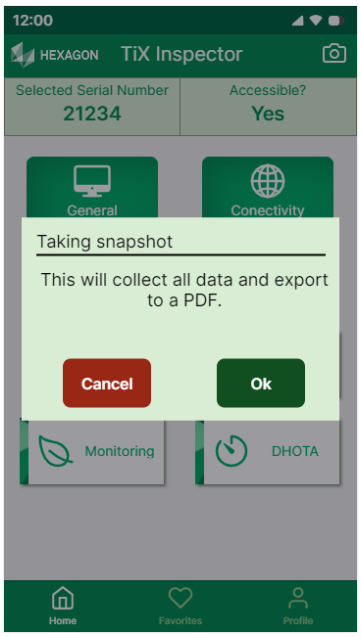
Fonte: Arquivo pessoal.

Figura 39 – Módulo “*Monitoring*”



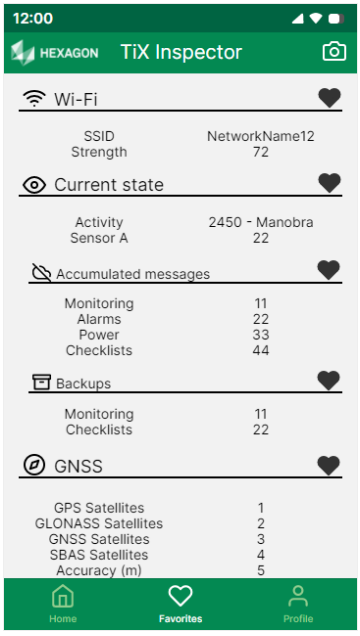
Fonte: Arquivo pessoal.

Figura 40 – Modal de confirmação



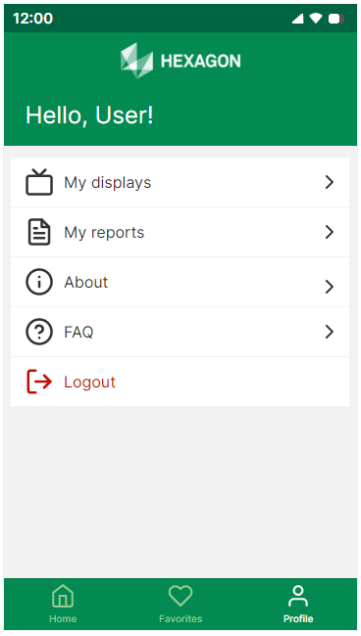
Fonte: Arquivo pessoal.

Figura 41 – Aba “*Favorites*”



Fonte: Arquivo pessoal.

Figura 42 – Aba “*Profile*”



Fonte: Arquivo pessoal.

APÊNDICE B – ESTRUTURA DO ESTADO DO APLICATIVO *MOBILE*

Figura 43 – Estrutura exemplo dos módulos

```
modules: [
    {
        id: 0,
        name: 'Módulo 0',
        iconName: 'ícone-do-módulo'
    },
    // ...
]

```

Fonte: Arquivo pessoal.

Figura 44 – Estrutura exemplo das seções

```
sections: [
    {
        id: 0,
        title: 'Seção 0',
        moduleId: 0,
        iconName: 'ícone-da-seção',
        isFavorite: true,
        data: [
            {
                type: 'lists',
                id: 0,
                isScrollable: true, // opcional
                height: 150 // opcional
            }
            // ...
        ]
    },
    // ...
]

```

Fonte: Arquivo pessoal.

Figura 45 – Estrutura exemplo das listas

```
lists: [
    {
        id: 0,
        name: "list_0",
        isLoading: false,
        labels: ["L1", "L2", "L3"],
        values: ["V1", "V2", "V3"],
    },
    // ...
]

```

Fonte: Arquivo pessoal.

Figura 46 – Estrutura exemplo das tabelas

```
tables: [
    {
        id: 0,
        name: "tabela_0",
        title: "Tabela 0", // opcional
        isLoading: false,
        header: ["H1", "H2", "H3"],
        rows: [
            ["C1", "C2", "C3"],
            // ...
            ["C4", "C5", "C6"]
        ],
    },
    // ...
]

```

Fonte: Arquivo pessoal.

Figura 47 – Estrutura do *login*

```
login: {
    isLogged: true,
    username: 'usuario',
    password: 'senha',
    name: 'Mateus Conceição',
    company: 'Hexagon',
    groups: [
        'Hexagon R&D',
        // ...
    ],
    tokens:  {
        accessToken: '',
        idToken: '',
        refreshToken: '',
    }
}

```

Fonte: Arquivo pessoal.

Figura 48 – Estrutura do restante do estado

```
displays: {
    registered: [
        1, 2, 3, 4,
        // ...
    ],
    selected: 1,
},
snapshotsFiles: [
    '1_snapshot_1600000000.pdf',
    // ...
],

```

Fonte: Arquivo pessoal.