

UNIVERSIDADE FEDERAL DE SANTA CATARINA

**EVOLUÇÃO DE UMA PLATAFORMA WEB PARA EXECUÇÃO DE APLICATIVOS
ANDROID DESENVOLVIDOS COM APP INVENTOR**

GUILHERME ALEXANDRINO PROENÇA
RAFAELA BERNARDO RODRIGUES

FLORIANÓPOLIS

2023/1

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE SISTEMAS DE INFORMAÇÃO

**EVOLUÇÃO DE UMA PLATAFORMA WEB PARA EXECUÇÃO DE APLICATIVOS
ANDROID DESENVOLVIDOS COM APP INVENTOR**

GUILHERME ALEXANDRINO PROENÇA
RAFAELA BERNARDO RODRIGUES

Trabalho de Conclusão de Curso de Graduação
em Sistemas de informação, do Departamento de
Informática e Estatística, do Centro Tecnológico
da Universidade Federal de Santa Catarina,
requisito parcial à obtenção do título de
Bacharel em Sistemas de informação.

Orientador: Prof. Jean Carlo Rossa Hauck, Dr.

FLORIANÓPOLIS

2023/1

RESUMO

A tecnologia e o uso de computadores e celulares estão cada vez mais presentes e desempenhando um papel fundamental no cotidiano da população. Entretanto, desenvolver um aplicativo para celular não é uma tarefa simples, pois é necessário ter conhecimento técnico em programação e possuir *softwares* específicos. Dessa forma, buscando romper com essa barreira técnica, começaram a surgir aplicações para facilitar o desenvolvimento de aplicativos e introduzir novos usuários à programação. Um exemplo disso é o *App Inventor*, uma ferramenta gratuita que tem como propósito promover o pensamento computacional por meio do desenvolvimento de aplicativos, utilizando programação com blocos visuais. No entanto, para se visualizar o aplicativo em tempo de desenvolvimento, o do *App Inventor* requer que o usuário possua um *smartphone* ou utilize um emulador, que, por sua vez, demanda um certo nível de capacidade do computador. Essa exigência pode se tornar uma restrição em escolas públicas onde muitos alunos não possuem celulares e os computadores são precários. Com isso, foi iniciado, como prova de conceito, o desenvolvimento de uma aplicação *web* para suprir a necessidade do uso do celular para conseguir visualização do andamento do projeto desenvolvido no *App Inventor*. Assim, este trabalho desenvolve novas funcionalidades como os componentes *ListView*, *Notifier*, *Web Viewer*, Câmera, Imagem, *Player*, *Sound e Imager Picker* que não foram implementadas.

Palavras-chave: Computação na escola, *App Inventor*. Android, plataforma *Web*

ABSTRACT

Technology and the use of computers and smartphones are becoming increasingly present and playing a fundamental role in people's daily lives. However, developing a mobile application is not a simple task, as it requires technical programming knowledge and specific software. In order to overcome this technical barrier, applications have started to emerge to facilitate app development and introduce new users to programming. An example of this is App Inventor, a free tool aimed at promoting computational thinking through app development using visual programming blocks. However, using App Inventor requires the user to have a smartphone or use an emulator, which in turn requires a certain level of computer capability to visualize the app during development. This requirement can become a restriction in public schools where many students do not have smartphones. As a proof of concept, the development of a web application was initiated to meet the need for using a mobile phone to visualize the progress of the project developed in App Inventor. Therefore, this work develops new functionalities such as ListView, Notifier, Web Viewer, Camera, Image, Player, Sound, and Image Picker components that have not been implemented.

Keywords: Computing in schools, App Inventor, Android, Web platform

LISTA DE ILUSTRAÇÕES

Figura 1. Os três eixos do pensamento computacional	16
Figura 2: Interface do <i>App Inventor</i> no modo de edição de interface	18
Figura 3: Interface do <i>App Inventor</i> no modo de edição de blocos	19
Figura 4: Estrutura do arquivo .aia	21
Figura 5: Conexão entre dispositivo móvel e computador com o <i>App Inventor</i> via Wi-Fi	23
Figura 6: Exemplo de código YAIL	24
Figura 7: Servidor <i>App Inventor</i> utiliza a GAE - <i>Google App Engine</i>	25
Figura 8: Interação <i>App Inventor</i> com Redis	32
Figura 9: Plataforma <i>Web</i> Criada para visualização de logs da CloudDB	33
Figura 10: Imagem do código do componente ListView	48
Figura 11: Imagem do código do componente Camcorder	52
Figura 12: Imagem do código do componente Camera	53
Figura 13: Imagem do código do componente <i>VideoPlayer</i>	52
Figura 14: Transmissão dos métodos do componente <i>VideoPlayer</i>	53
Figura 15. Trecho do código fonte do método putAsset do App Inventor	54
Figura 16: Trecho do código fonte do método put da plataforma LIDEA-W	55
Figura 17: Imagem do código do componente Player	56
Figura 18: Transmissão dos métodos do componente Player	57
Figura 19: Trecho de código do componente <i>Image Picker</i>	58
Figura 20: Trecho de código da função <i>updateMedias</i>	59
Figura 21: Trecho de código da inicialização do indexedDB	60
Figura 22: Trecho de código do Context do TinyDB	61
Figura 23: Exemplo de sintaxe de condicionais no YAIL	61
Figura 24: Exemplo de algumas palavras-chave no dicionário para traduzir o YAIL	62
Figura 25: Alguma das melhorias na tradução de condicionais do YAIL	63
Figura 26: Método responsável por alterar a visibilidade dos componentes	64
Figura 27: Método <i>updateProperties</i> sendo emitido	64

Figura 28: Imagem do aplicativo no App Inventor	66
Figura 29: Imagem dos blocos do aplicativo no App Inventor	67
Figura 30: Imagem do aplicativo na aplicação LIDEA-W	68
Figura 31: Imagem do ListView no App Inventor	68
Figura 32: Imagem da inicializacao da variavel global	69
Figura 33: Imagem da inicialização do ListView1	69
Figura 34: Imagem da inicialização do Button1	70
Figura 35: Imagem da aplicação	71
Figura 36: Imagem das propriedades do Player	72
Figura 37: Imagem do componente Sound	72
Figura 38: Imagem das as telas do VideoPlayer e Sound	73
Figura 39: Imagem dos blocos das funções básicas do VideoPlayer e Sound	74
Figura 40: Imagem dos blocos usados no teste do componente Notifier	75
Figura 41: Imagens do resultado do componente Notifier no LIDEA-W	75
Figura 42: Resultado do Webviewer	76
Figura 43: Blocos do componente Camera	77
Figura 44: Resultado do componente de Camera na aplicação LIDEA-W	78
Figura 45: Blocos do componente Camcorder	78
Figura 46: Exibição do Video na aplicação LIDEA-W	79
Figura 47: Blocos de IF/else no App inventor	80
Figura 48: Resultado dos blocos de IF/else no LIDEA-W	80

LISTA DE TABELAS

Tabela 1: Termos utilizados nas buscas	29
Tabela 2: Primeira string de busca utilizada na pesquisa	30
Tabela 3: Segunda string de busca utilizada na pesquisa	30
Tabela 4: Terceira string de busca utilizada na pesquisa	30
Tabela 5: Quarta e última string de busca utilizada na pesquisa	31
Tabela 6: Requisito funcional 1	35
Tabela 7: Requisito funcional 2	36
Tabela 8: Requisito funcional 3	37
Tabela 9: Requisito funcional 4	38
Tabela 10: Requisito funcional 5	38
Tabela 11: Requisito funcional 6	39
Tabela 12: Requisito funcional 7	40
Tabela 13: Requisito funcional 8	40
Tabela 14: Requisito funcional 9	40
Tabela 15: Requisito funcional 10	41
Tabela 16: Requisito funcional 11	42
Tabela 17: Requisito funcional 12	42
Tabela 18: Requisito funcional 13	43
Tabela 19: Requisito funcional 14	43
Tabela 20: Tabela de definição dos pontos de história	44
Tabela 21: Tabela de definição dos pontos de história para cada requisito funcional	45
Tabela 22: Tabela de apresentação das sprints por requisito funcional	46
Tabela 23: Tabela 23: Resumo dos resultados dos testes realizados	81

LISTA DE ABREVIACOES E SIGLAS

AJAX	<i>Asynchronous JavaScript</i>
API	<i>Application Programming Interface</i>
DOM	<i>Document Object Model</i>
GAE	<i>Google App Engine</i>
GWT	<i>Google Web Toolkit</i>
INEP	Instituto Nacional de Estudos e Pesquisas
IP	<i>Internet Protocol</i>
JSON	<i>JavaScript Object Notation</i>
JVM	<i>Java Virtual Machine</i>
MIT	<i>Massachusetts Institute of Technology</i>
RPC	<i>Remote Procedure Call</i>
YAIL	<i>Young Android Intermediate Language</i>
XML	<i>Extensible Markup Language</i>

SUMÁRIO

1 Introdução	11
1.1 Objetivos	12
1.1.1 Objetivo geral	12
1.1.2 Objetivos específicos	13
2 Fundamentação Teórica	15
2.1 Ensino de computação	15
2.1.1 Computação na escola	16
2.2 App Inventor	17
2.2.1 MIT AI2 Companion	21
2.2.2 YAIL	23
2.2.3 GWT	24
2.3 LIDEA-W	25
2.3.1 React	25
2.3.2 NodeJS	26
3 Trabalhos correlatos	27
3.1 Plataforma Web para execução de apps Android desenvolvidos com App Inventor	27
3.2 Atualização do estado da arte	28
3.2.1 Protocolo de pesquisa	29
3.2.2 Execução da pesquisa	30
3.2.3 Análise do resultados	31
3.3 Considerações Finais	34
4 Análise e projeto	35
4.1 Levantamento de análise de requisitos	35
4.1.2 Requisitos funcionais	35
4.1.2.1 Componentes de interface do usuario	35
4.1.2.2 Componentes de mídia	38
4.1.2.3 Componentes de armazenamento	42
4.1.2.4 Implementações na lógica de interpretação do código Yail	42
4.2 Estimativa	44
5 Implementação	46
5.1 Planejamento	46
5.2 Desenvolvimento	47
5.2.1 ListView	48
5.2.2 Notifier	49
5.2.3 WebViewer	49
5.2.4 Camcorder	50
5.2.5 Camera	51
5.2.6 MediaPlayer	52
5.2.7 Image	53
5.2.8 Player	56

	10
5.2.9 Sound	57
5.2.10 Image Picker	58
5.2.11 TinyDb	60
5.2.12 Refatoração das condicionais	61
5.2.13 Alterar visibilidade dos componentes	63
5.2.14 Refatoração variáveis globais	65
5.3 Testes Funcionais	65
5.4 Validação dos Requisitos	81
6 Conclusão	83
6.1 Trabalhos Futuros	84
Referências	85

1 Introdução

O *App Inventor* é uma aplicação web de código aberto desenvolvida pelo *Massachusetts Institute of Technology (MIT)* (MIT APP INVENTOR, 2022). Seu propósito é promover o pensamento computacional por meio do desenvolvimento de aplicativos móveis amadores, sem a necessidade de se aprender uma linguagem textual, pois utiliza programação com blocos visuais. Dessa forma, a ferramenta *App Inventor* democratiza o aprendizado do pensamento computacional e estimula o interesse de crianças e adolescentes na área de tecnologia. Com mais de um milhão de visitantes mensais de 195 países diferentes (MIT APP INVENTOR, 2022), a aplicação passou a ser usada em inúmeros projetos e iniciativas que buscam inserir o ensino da computação na educação básica em diversos países e no contexto da educação brasileira não foi diferente.

Atualmente, a depuração dos aplicativos desenvolvidos com o *App Inventor* necessita do suporte de um aplicativo chamado *MIT AI2 Companion*. Através desse app é possível realizar a conexão a um dispositivo (smartphone ou tablet) que executará o aplicativo em desenvolvimento. A conexão entre o aplicativo *MIT AI2 Companion* e o *App Inventor* pode ser feita via Wi-Fi, no entanto ambos devem estar conectados à mesma rede local. No entanto, o Censo Escolar do ano de 2021, realizado pelo INEP, indicou que menos de 40% das escolas do ensino fundamental da rede pública possuem acesso a um computador e acesso à internet para ensino e aprendizagem (INEP, 2021).

Considerando então o processo de conexão para a etapa de depuração e o cenário atual do resultado da pesquisa feita pelo INEP, tem-se uma limitação de uma das etapas fundamentais para quem está iniciando a prática do desenvolvimento que é visualizar, testar e entender o que foi criado. Além da falta de computadores com o acesso a internet, a necessidade de um smartphone é um agravante pois não faz parte da realidade de muitas das crianças e adolescentes de escola pública como tem sido percebido nas aplicações voltadas ao ensino do pensamento computacional realizadas pela iniciativa Computação na Escola¹ no Brasil.

A Iniciativa Computação na Escola, do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina, tem como objetivos: motivar e facilitar o ensino da computação para todos os alunos do Ensino Fundamental e Médio no Brasil, capacitar professores para esse ensino, compartilhar relatos de experiências e recursos a fim de criar uma comunidade de prática e aumentar o número de formados na área de computação - buscando elevar a representatividade de mulheres e minorias (COMPUTAÇÃO NA

¹ <https://computacaonaescola.ufsc.br/>

ESCOLA, 2022). A Iniciativa tem realizado diversos tipos de eventos com escolas parceiras visando o ensino da computação, fornecendo à comunidade oficinas de aprendizado e exposições sobre o tema, além de manter em seu site cursos gratuitos² como: *Machine Learning*, Princípios do design visual de interface, entre outros.

Para tentar diminuir a complexidade e as diversas limitações para a utilização do *App Inventor*, foi iniciado o desenvolvimento de uma ferramenta *web*, chamada LIDEA-W, para a visualização do aplicativo em desenvolvimento e sua depuração sem a necessidade de criação de uma máquina virtual, utilização de emuladores ou utilização de um *smartphone* (BRANDT, 2021).

Esse projeto começou em 2021 desenvolvido pelo bacharel em Sistemas da Informação Darlan Brandt (BRANDT, 2021), que implementou o protocolo de comunicação utilizado entre a aplicação *App Inventor* ao aplicativo *AI2 Companion*, implementou a comunicação com o servidor *Rendezvous*, simulando a utilização do aplicativo *AI2 Companion*, e iniciou o desenvolvimento da aplicação *web* que exibe, em tempo de desenvolvimento, na tela do navegador o aplicativo construído dentro do *App Inventor*. Esse trabalho resultou em um primeiro protótipo que serviu como prova de conceito da viabilidade de implementação dessa alternativa de conexão.

No entanto, o trabalho de Brandt (2021) propõe diversas melhorias, que não puderam ser implementadas por questão de complexidade e tempo, como por exemplo alguns componentes não visuais do *App Inventor* que podem ser convertidos em novos componentes, pois podem utilizar *hardware* disponível em computadores, como microfone, caixas de som e câmera. A inclusão de tais componentes acrescenta funcionalidades que tornariam o ensino do uso da ferramenta mais divertido para os estudantes (BRANDT, 2021).

Com isso, este trabalho propõe a evolução desta aplicação para que seja possível a implantação do sistema *web* na nuvem de forma a atenuar as dificuldades da utilização do *App Inventor*, sobretudo em escolas públicas com alunos de baixa renda que podem não possuir acesso a *smartphones*.

1.1 Objetivos

1.1.1 Objetivo geral

² <https://computacaonaescola.ufsc.br/crie-apps/>

Este trabalho tem como objetivo principal a evolução da ferramenta *web* LIDEA-W de execução de aplicativos desenvolvidos com *App Inventor* por meio do desenvolvimento de componentes atualmente não suportados pela ferramenta.

1.1.2 Objetivos específicos

Para que seja possível alcançar o objetivo geral, os seguintes objetivos específicos devem ser cumpridos:

- Atualizar a análise do estado da arte em relação à implementação de emuladores Android para web em geral e especificamente para *App Inventor*
- Analisar as melhorias necessárias na implementação atual da ferramenta LIDEA-W, para que sejam elencadas as melhorias de padrão de projetos necessárias e novas funcionalidades a serem suportadas
- Analisar o protocolo de comunicação entre a aplicação *App Inventor* e o aplicativo *AI2 Companion* para que seja possível implementar as novas funcionalidades
- Desenvolver e testar as novas funcionalidades para evolução da LIDEA-W

1.2 Metodologia

Esta seção apresenta como será executada a pesquisa e a estrutura metodológica que se pretende adotar: Este trabalho pode ser caracterizado como pesquisa aplicada pois visa gerar conhecimentos para aplicação prática dirigidos à solução de problemas específicos (SILVA; MENEZES, 2005). Então, para que se alcance uma solução que englobe os objetivos específicos do projeto, são estabelecidas etapas de procedimentos metodológicos da seguinte forma:

Fase 1: Análise do Estado da Arte:

A1.1 - Definir o protocolo de pesquisa

A1.2 - Atualizar os estudos com conteúdo relevante ao tema do trabalho

A1.3 - Selecionar e filtrar os estudos que mais se aproximam do tema do trabalho

A1.4 - Analisar os resultados obtidos

A1.5 Estudo da fundamentação teórica: Estudo teórico sobre o funcionamento da comunicação do *App Inventor* com *AI2 Companion* para que seja possível desenvolver as novas funcionalidades

Fase 2: Estudo da ferramenta atual:

A2.1: Configuração do ambiente necessário para rodar a plataforma LIDEA-W: Etapa destinada à instalação e configuração do ambiente de desenvolvimento necessário para execução da plataforma LIDEA-W localmente.

A2.2: Análise do código da plataforma LIDEA-W: Nesta etapa serão feitos testes e análise do que pode ser melhorado na plataforma a fim de cumprir o objetivo geral deste trabalho.

Fase 3: Levantamento e análise dos requisitos:

A3.1: Levantamento inicial dos requisitos: Criar os requisitos funcionais para o desenvolvimento das novas funcionalidades elencadas da análise feita na etapa anterior.

Fase 4: Implementação e testes:

A4.1: Desenvolvimento da solução: Implementação dos requisitos funcionais listados na etapa anterior.

A4.2: Realização de testes dos componentes implementados a partir dos requisitos funcionais.

2 Fundamentação Teórica

Neste capítulo é apresentada a fundamentação teórica sobre os assuntos envolvidos neste trabalho. São abordados conceitos básicos sobre o ensino de computação na educação, as tecnologias envolvidas no desenvolvimento do *App Inventor* e as tecnologias utilizadas para o desenvolvimento da ferramenta LIDEA-W.

2.1 Ensino de computação

O Ministério da Educação do Brasil homologou as diretrizes para o ensino de computação na educação básica de todo o país. A normatização foi elaborada pelo Conselho Nacional de Educação (CNE) e atende ao art.22 da Resolução CNE nº 2, de 22 de dezembro de 2017, que instituiu e orientou a implantação da Base Nacional Comum Curricular (BNCC). (Brasil, 2022).

Com o respaldo da Sociedade Brasileira de Computação (SBC), foram desenvolvidas diretrizes para o ensino de Computação que abordam as competências e habilidades relacionadas aos seguintes eixos: Pensamento Computacional, Mundo Digital e Cultura Digital (SBC, 2019). Essas diretrizes foram incorporadas à Base Nacional Comum Curricular (BNCC), abrangendo desde a Educação Infantil até o Ensino Médio:

- Pensamento computacional: Refere a capacidade de compreender, analisar e abstrair problemas utilizando técnicas necessárias para a descrição e análise de informações e processos, bem como para a automação de soluções (SBC, 2019).
- Mundo Digital: Compreende artefatos digitais – físicos (computadores, celulares, tablets) e virtuais (internet, redes sociais, programas, nuvens de dados). Mundo digital diz respeito à informação, armazenamento, proteção, e uso de códigos para representar diferentes tipos de informação, formas de processar, transmitir e distribuí-la de maneira segura e confiável (BRACKMANN, [s.d.]).
- Cultura Digital: Compreender o impacto e decorrências da revolução digital e dos avanços do mundo digital na humanidade. Utilizando de forma eficiente e crítica ferramentas que auxiliem a obter, analisar, sintetizar e comunicar informações de formatos e com fins diversos e analisar de forma crítica questões éticas e morais que surgiram com o mundo digital (SBC, 2017).



Figura 1: Os três eixos do pensamento computacional (BRACKMANN, 2022)

Apesar da homologação das diretrizes do ensino de computação na educação ser um grande passo para a educação brasileira, em diversos países do mundo já consideram o ensino da computação no currículo escolar e diversas diretrizes já existem. Um exemplo é o *K–12 Computer Science Framework*, desenvolvido pela *Computer Science Teachers Association*, que atualmente é a maior referência internacional de *framework* de currículo escolar para o ensino de computação. Um dos objetivos do *framework* é introduzir os conceitos fundamentais da ciência da computação para todos os alunos, começando no nível do ensino fundamental e apresentando ciência da computação no nível da escola secundária de uma forma que possa facilitar a inclusão em cursos de graduação em ciência da computação, matemática ou ciências. (COSTA, 2019)

2.1.1 Computação na escola

As mudanças políticas, econômicas e sociais que vêm ocorrendo nos últimos anos são responsáveis por alterar de forma significativa a sociedade. Fenômenos como a Globalização e a chamada Era da Informática introduziram novos desafios à sociedade atual (ANTONIO e LEMOS, 2015). Esses fenômenos contribuem para a evolução da tecnologia e seus dispositivos tecnológicos, além de também alterarem a forma como o conhecimento é encarado, produzido e transmitido.

Nesse cenário de evolução tecnológica, é normal que a demanda por profissionais capacitados também aumente e que estes devam possuir habilidades relacionadas ao pensamento computacional, mesmo que não trabalhem em áreas ligadas diretamente à computação. Entretanto, é importante que esse conhecimento seja desenvolvido o quanto antes, para facilitar a familiarização ao conhecimento computacional e o aprendizado de novas formas de resolver problemas.

De acordo com GARLET et al. (2016), no Brasil o aprendizado da Informática ou mais especificamente da programação, está quase restrito apenas às pessoas que buscam por essas áreas de conhecimento. Entretanto, surgem iniciativas como o Computação na Escola, com o intuito de alterar esse cenário.

2.2 App Inventor

O *App Inventor* é uma ferramenta de programação visual baseada em blocos, que possibilita a criação de aplicativos para sistemas Android por meio de uma interface acessada via navegador *web*. Lançado pela Google em 2010, atualmente é mantido pelo MIT (*Massachusetts Institute of Technology*). Seu principal objetivo é democratizar o desenvolvimento de *software*, capacitando todas as pessoas, especialmente os jovens, a transitar do consumo para a criação de tecnologia.

De acordo com Lye E Koh (2014) a democratização do *App Inventor* se deve porque ambientes de programação baseados em blocos simplificam a programação, já que se fundamentam em um processo de arrastar e encaixar blocos de comando. Isso reduz as ações de programar em si, fazendo com que os estudantes foquem na lógica, sem se preocupar com a sintaxe da programação, assim facilitando o aprendizado. O modo de edição visual é demonstrado na figura 2.

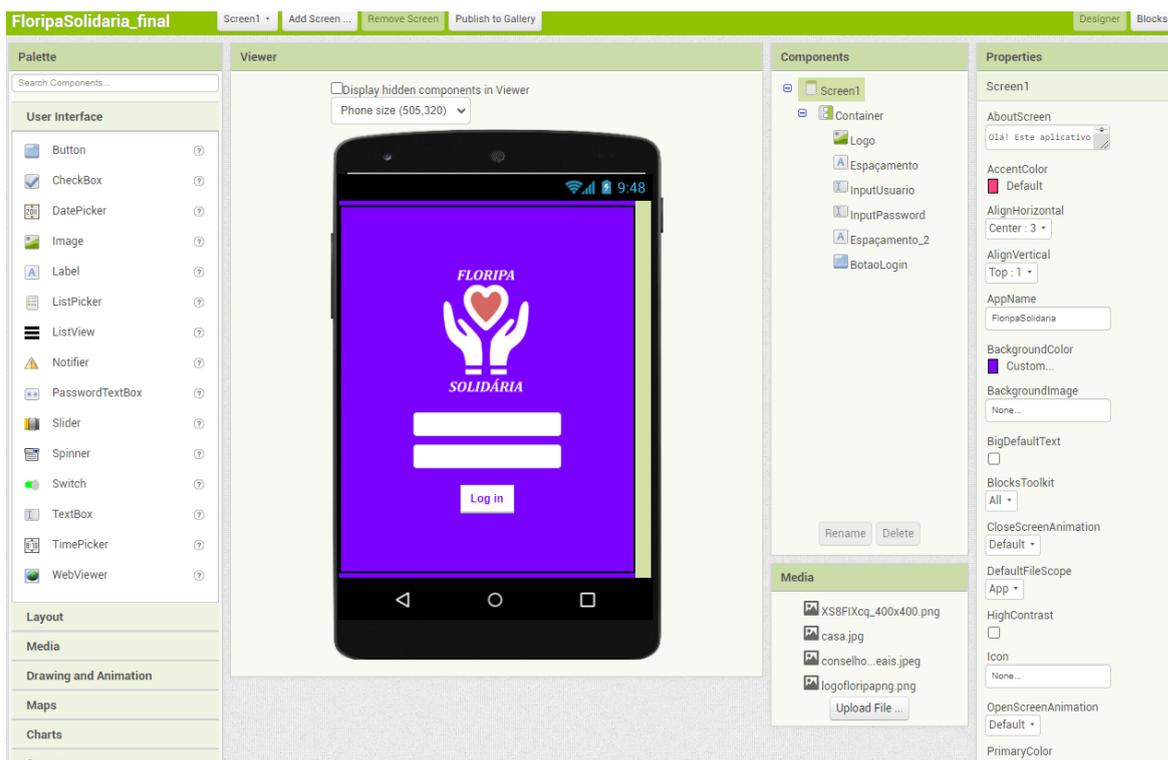


Figura 2: Interface do *App Inventor* no modo de edição de interface (Elaborado pelos autores)

A programação em blocos tem ganhado destaque nos últimos anos, de acordo com Gonçalves (2022). Isso se deve à sua metodologia, que dispensa o uso de uma linguagem de programação convencional, eliminando a necessidade de conhecimento de sintaxe e regras. Em vez de linhas de código, são utilizados blocos com formas e cores específicas, que auxiliam na diferenciação de suas funções. A organização sequencial desses blocos possibilita a realização de tarefas e a resolução de problemas.

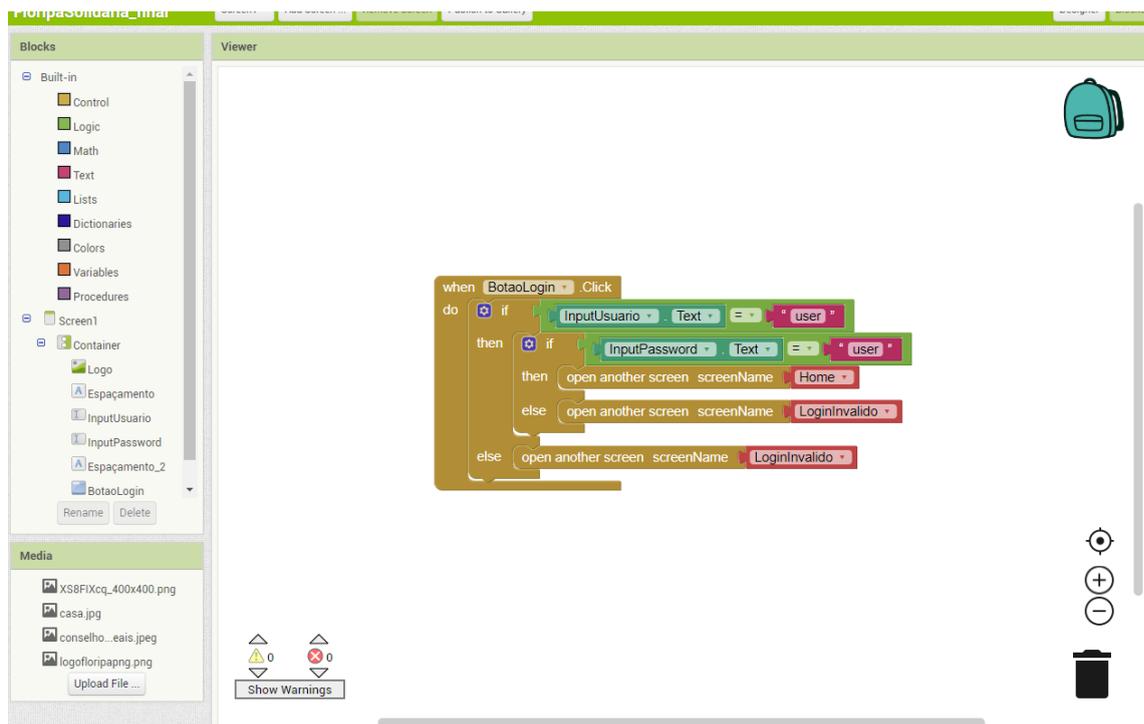


Figura 3: Interface do *App Inventor* no modo de edição de blocos (Elaborado pelos autores)

A ferramenta do *App Inventor* é composta basicamente por dois módulos: cliente e servidor. O módulo cliente corresponde a interface de desenvolvimento da aplicação, ou seja, onde o usuário escreve o código e cria a interface - onde são incluídos botões, campos, sensores, entre outros. O módulo servidor contém a parte responsável pela autenticação e salva no banco de dados os projetos criados pelo usuário (Dominguez e Spertus, 2022). De acordo com GARCIA (2002) a aplicação também possui componentes de comunicação compartilhados entre os módulos, essa comunicação é realizada pelo protocolo RPC (*Remote Procedure Call*), o qual executa de forma remota procedimentos em componentes ligados em uma rede.

Conforme a documentação do *App Inventor* descrita por Domingues e Spertus (2022), no que tange a parte de um projeto e sua organização, a distribuição da aplicação consiste em nove subdiretórios, dos quais os sete primeiros contém código-fonte para subprojetos e os últimos dois contém arquivos estáticos. A seguir a estrutura de diretórios é apresentada:

- *aiphoneapp*: o interpretador executado no smartphone ou emulador quando conectado a um computador onde o *App Inventor* está sendo utilizado;
- *Aiplayapp*: interpretador referente ao *AI2 Companion*, executado no smartphone ou emulador quando conectado a um computador onde o *App Inventor* está sendo utilizado;

- *Appengine*: é o aplicativo GWT o qual fornece código JavaScript para o lado do navegador do cliente, além de oferecer suporte à funcionalidade do lado servidor que salva e recupera os projetos e emite solicitações de compilação para o servidor;
- *Blocklyeditor*: o editor de blocos em si, está presente no navegador (lado cliente) e faz uso da biblioteca Blockly;
- *Buildserver*: é um servidor http que utiliza um arquivo zip e produz um apk, ou caso não consiga, produz uma mensagem de erro;
- *Common*: constantes e classes utilitárias utilizadas por outros subprojetos;
- *Components*: código para componentes do *App Inventor*, inclui anotações, implementações e scripts para extrair informações de componentes necessários para o subprojeto;
- Docs: documentação de usuário, como por exemplo: tutoriais;
- Lib: bibliotecas externas;

A plataforma do *App Inventor* traz funcionalidades como importação e exportação, para facilitar o uso. Ao utilizar a função de exportar um projeto é gerado um arquivo no formato .aia, que contém todas as informações da aplicação criada. Conforme BRANDT (2021), o código do app fica dentro do diretório src do arquivo e os códigos são separados em dois formatos principais: .scm ou .bky. No primeiro estão listadas as informações do layout da tela, em formato JSON, enquanto no segundo estão listadas as informações da lógica em formato XML. A figura 4 apresenta a estrutura do arquivo.

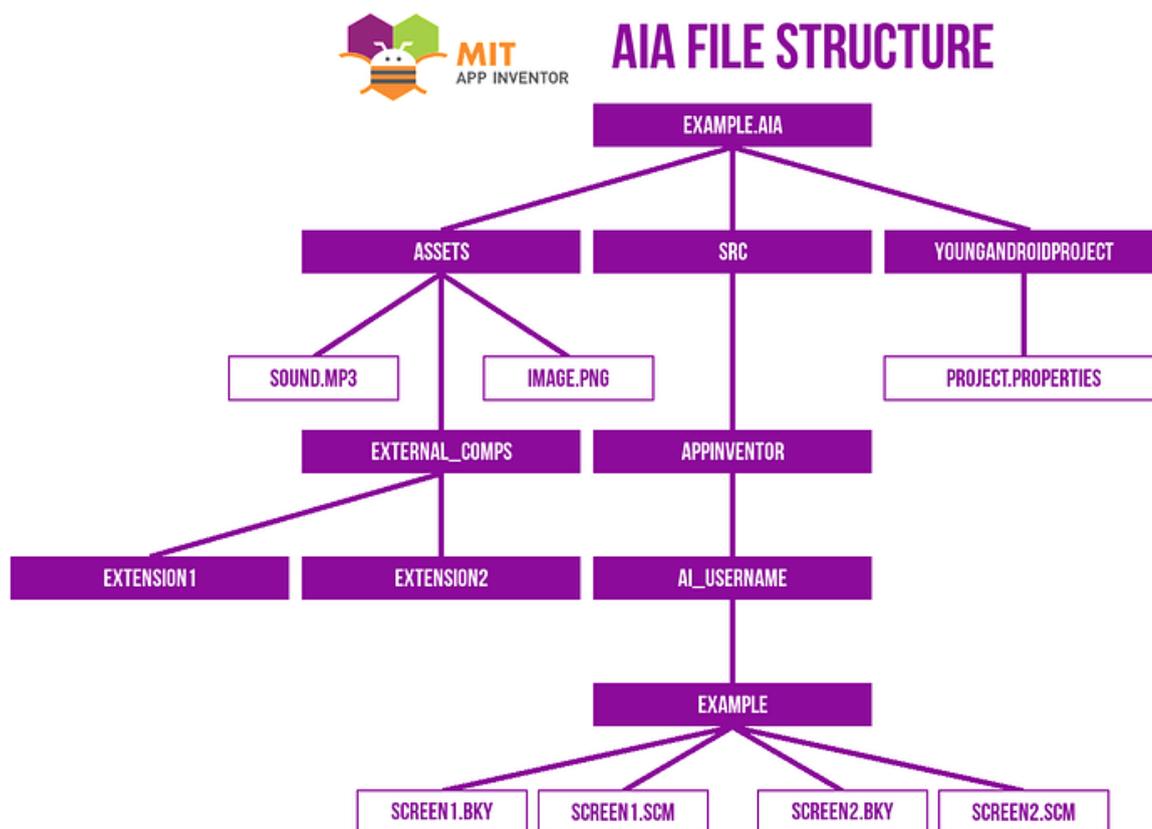


Figura 4: Estrutura do arquivo .aia (MIT APP INVENTOR COMMUNITY, 2019)

Conforme FRANÇA (2019), dentro do *App Inventor* existem algumas formas de realizar o *live testing* do aplicativo desenvolvido, uma delas é por meio do MIT *AI2 Companion*, que consegue realizar o debugging da aplicação desenvolvida em tempo real, em um celular. Via plataforma do *App Inventor*, a aplicação desenvolvida também pode ser exportada para um arquivo .apk, que pode ser distribuído através da *Play Store* ou instalado em smartphones com sistema Android.

2.2.1 MIT AI2 Companion

O *MIT AI2 Companion* é um aplicativo para Android e IOS que permite realizar a conexão entre um *smartphone* ou *tablet* via USB ou via Wi-Fi com a aplicação desenvolvida no *App Inventor*, com o objetivo do usuário conseguir acompanhar a evolução do seu aplicativo desenvolvido no *App Inventor*. Ou seja, o usuário irá conseguir executar o código desenvolvido e com isso testar sua aplicação, visualizando o resultado em tempo real, em seu *smartphone* ou *tablet*.

Segundo SCHILLER, et al. (2014) a principal função de visualizar o aplicativo desenvolvido em tempo real, seria o *live debugging*, que permite obter uma forma de

visualizar imediatamente as alterações realizadas no programa desenvolvido, assim, o programador consegue entender e experimentar o aplicativo criado. Conforme FRANÇA (2019), vale lembrar que o usuário deve ter um aparelho Android ou IOS com o *Companion* instalado e deve estar conectado na mesma rede wi-fi que o computador onde a aplicação está sendo desenvolvida.

A conexão entre o smartphone e um computador só é possível através da conexão com o MIT RendezVous *Server*, e tanto o smartphone quanto o computador necessitam estar conectados na mesma rede. A conexão é realizada em seis passos (BRANDT, 2021):

1. Computador faz *check-in* com um código de seis caracteres gerado pelo *MIT AI2 Companion*;
2. MIT RendezVous *Server* registra o endereço IP do computador;
3. Dispositivo móvel faz *check-in* com um código de 6 caracteres;
4. MIT RendezVous *Server* registra o endereço IP do dispositivo móvel;
5. MIT RendezVous *Server* informa os endereços IP dispositivo móvel e o computador, que possuem a mesma chave de seis caracteres, para o outro;
6. Conexão estabelecida e ambos passam a se comunicar via IP.

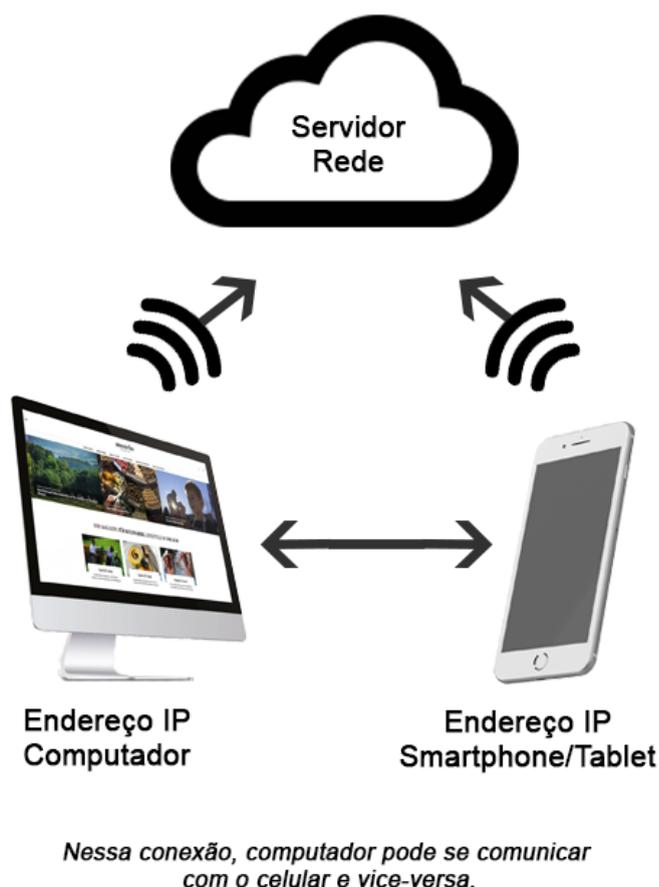


Figura 5: Conexão entre dispositivo móvel e computador com o *App Inventor* via Wi-Fi (Elaborado pelos autores)

Segundo SCHILLER et al. (2014), o envio das chaves de seis caracteres para o servidor RendezVous não é feito em texto simples, todas comunicações são feitas de forma segura. Nesse caso utilizando o algoritmo de hash MD5. O computador, ao se conectar ao dispositivo móvel, funciona como um servidor *web* - o qual envia código YAIL (implementação de uma outra linguagem de programação KAWA) com uma chamada AJAX.

O *Companion* é responsável por checar se houve modificação no código ou não. Caso tenham ocorrido mudanças, ele exibe as alterações em tempo real na tela. Para conseguir realizar tal tarefa ele verifica se o código recebido é igual ao que está em cache, se for diferente ele atualiza.

2.2.2 YAIL

Uma das etapas para criar um aplicativo no *App Inventor* é a conversão dos componentes e blocos de todas as telas contidas na aplicação para uma linguagem intermediária chamada de *Young Android Intermediate Language* (YAIL), uma linguagem

definida como uma coleção de macros e funções em Kawa, uma implementação baseada em Java da linguagem funcional Scheme (SCHILLER et al., 2014).

```

;;; Screen1
(do-after-form-creation
  (set-and-coerce-property! 'Screen1 'Title "Screen1" 'text))

;;; Canvas1
(add-component Screen1 Canvas Canvas1
  (set-and-coerce-property! 'Canvas1 'BackgroundColor
    #xFF00FFFF 'number)
  (set-and-coerce-property! 'Canvas1 'Width 200 'number)
  (set-and-coerce-property! 'Canvas1 'Height 300 'number))

;;; Ball1
(add-component Canvas1 Ball Ball1
  (set-and-coerce-property! 'Ball1 'X 46 'number)
  (set-and-coerce-property! 'Ball1 'Y 27 'number))

(define-event Ball1 Flung($x $y $speed $heading $xvel $yvel)
  (set-this-form)
  (set-and-coerce-property! 'Ball1 'Speed
    (lexical-value $speed) 'number)
  (set-and-coerce-property! 'Ball1 'Heading
    (lexical-value $heading) 'number))

```

Figura 6: Exemplo de código YAIL (SCHILLER et al., 2014)

Já a linguagem Kawa roda na plataforma Java. Ela visa combinar os benefícios das linguagens de script dinâmicas com os benefícios das linguagens compiladas tradicionais (GNU, [s.d.]). Além disso, a linguagem KAWA é importante no contexto do *App Inventor* pois possibilita integrar bibliotecas Java e compilar em *Java Virtual Machine* (JVM), que em tempo de execução cria as classes Java apropriadas para representar as telas e componentes em um aplicativo (SAYYAPARAJU, 2014).

2.2.3 GWT

Google Web Toolkit (GWT) consiste em um kit de ferramentas para criar e otimizar aplicativos complexos. Tem como objetivo aumentar a produtividade em aplicativos *web* de alto desempenho sem que o desenvolvedor precise ser um especialista em peculiaridades do navegador, XMLHttpRequest e JavaScript. É uma ferramenta de código aberto e totalmente gratuita (GWT PROJECT, 2022).

O SDK (*Software Development Kit*) do GWT contém as bibliotecas JAVA, o compilador e o servidor de desenvolvimento. Com ele é possível desenvolver aplicações em JAVA e implantá-las como JavaScript. Com o GWT é possível escrever aplicativos altamente

otimizados em JavaScript, os quais funcionam em todos os navegadores, incluindo navegadores móveis para Android e iOS (GWT PROJECT, 2022).

Utilizando o GWT os programadores não precisam se preocupar com os detalhes das chamadas RPC (Remote Procedure Calls), exceto para fornecer retornos (callbacks) bem-sucedidos ou com falhas. (DOMINGUEZ e SPERTUS, 2022).

Conforme DOMINGUEZ e SPERTUS (2022), o GWT funciona muito bem com a Google App Engine (GAE). GAE é uma plataforma de computação em nuvem que permite que programas escritos em JAVA ou Python sejam armazenados e executados nos servidores da Google.



Figura 7: Servidor *App Inventor* utiliza a GAE - *Google App Engine* (DOMINGUEZ e SPERTUS, 2022)

Na figura 7 é possível observar que o cliente e servidor do *App Inventor* são criados utilizando GWT, o qual é convertido no *frontend* para JavaScript, sendo executado no navegador do usuário. Por sua vez, o *backend* não é convertido, é executado na biblioteca do servidor GWT como um serviço do *Google App Engine*.

2.3 LIDEA-W

Nesta seção são abordados aspectos teóricos das tecnologias utilizadas na plataforma web LIDEA-w que foi desenvolvida por Brandt (2021) com o objetivo de criar um sistema web para a execução, de forma funcional, de telas de aplicativos móveis desenvolvidos na ferramenta *App Inventor*, respeitando a restrição de projeto imposta que dispõe-se a preservar as escolhas do autor em relação às linguagens e ferramentas de desenvolvimentos aplicadas na elaboração da aplicação.

2.3.1 React

React é uma biblioteca de código aberto escrita em JavaScript para construção de interfaces de usuário. Ela foi desenvolvida pela empresa Meta (FACEBOOK INC., 2021) e está disponível desde 2013. A biblioteca é mantida pela mesma empresa em colaboração com

a comunidade de desenvolvedores, tendo recentemente lançado a versão 18. A adoção do React começou a se expandir em 2015, quando a empresa Netflix anunciou o uso da biblioteca para potencializar o desenvolvimento de interfaces gráficas de suas aplicações. Além disso, pelo quinto ano consecutivo, o React foi eleito a biblioteca mais utilizada para desenvolvimento web, de acordo com a pesquisa *Developer Survey* conduzida pela plataforma *Stack Overflow* em 2022

Um dos motivos da popularidade se deu pelo uso do JSX que é uma sintaxe baseada em tags na qual combina a linguagem javascript e XML que permite utilizar os elementos React e assim fornecer uma sintaxe concisa para criar elementos complexos na árvore DOM com atributos (BANKS et al., 2020). Com isso foi possível ter uma leitura agradável do código e familiaridade com HTML que já era muito conhecido pela comunidade de desenvolvedores.

Além disso, a biblioteca utiliza de recursos que garantem um ganho no desempenho como por exemplo, a virtualização do DOM. A biblioteca cria um cache de estrutura de dados na memória, quando esse DOM virtual recebe uma atualização a partir da interação do usuário, react salva um objeto da versão antes da alteração e a utiliza para calcular as diferenças e encontrar exatamente onde foram realizadas as modificações. Dessa forma a biblioteca consegue atualizar o DOM real apenas onde é necessário, salvando o esforço de recalcular a página inteira e suas dependências (CODECADEMY, 2022).

2.3.2 NodeJS

O *backend* da aplicação utiliza Node.js que foi inicialmente desenvolvido por Ryan Dahl baseado no mecanismo JavaScript V8 do navegador Chrome do Google. Ele é um ambiente de tempo de execução JavaScript multiplataforma, gratuito e de código aberto que permite aos desenvolvedores escrever ferramentas de linha de comando e scripts do lado do servidor fora de um navegador (NODE.JS, 2022).

O Node.js conquistou grande popularidade nos últimos anos. De acordo com o relatório da Node.js Foundation, em 2018 foram realizados mais de 1 bilhão de downloads. Ele se destaca como um framework desenvolvido para otimizar a taxa de transferência e a escalabilidade de aplicativos da web, sendo uma solução eficaz para diversos desafios comuns no desenvolvimento web. No contexto do LIDEA-w, o Node.js é utilizado como uma arquitetura de microservices, que consiste em uma coleção de pequenos serviços autônomos. Cada serviço é independente e responsável por implementar uma única funcionalidade em um

contexto limitado. Sua utilização ocorre por meio de APIs, que encaminham as chamadas para os serviços apropriados no *backend* (MICROSOFT, [s.d.])."

3 Trabalhos correlatos

Neste capítulo é apresentado o estado da arte visando a atualização das abordagens para realizar *live debugging* para *App Inventor*. Inicialmente é apresentada a monografia que originou a pesquisa que este trabalho se propõe evoluir. Na sequência, o levantamento do estado da arte realizado no trabalho original é atualizado por meio de um Mapeamento Sistemático da Literatura que visa identificar, classificar e interpretar pesquisas disponíveis por meio de critérios de qualificação claros e reproduzíveis em relação ao tema deste trabalho (PETERSEN et al., 2008)

3.1 Plataforma Web para execução de apps Android desenvolvidos com App Inventor

O trabalho de Brandt (2021) descreve a arquitetura e implementação da aplicação chamada *Web Live Debugging for App Inventor* (LIDEA-w) que tem como objetivo suportar a conexão com a ferramenta *App Inventor*, permitindo a exibição funcional dos componentes visuais e de layout definidos em tempo de desenvolvimento (BRANDT, 2021).

Brandt (2021) inicialmente realiza um estudo do código fonte do *App Inventor* a fim de investigar como funcionava a interação dos protocolos e tecnologias envolvidas na comunicação do *App Inventor* com o *MIT AI2 Companion*. Nessa etapa foi concluído que o arquivo responsável pela comunicação se chama *replmgr.js*, onde estão os comandos para geração do QR Code e da chave de seis caracteres para estabelecer a conexão (BRANDT, 2021). Ao identificar a função que realiza a conexão, foi criado um *log* para mapear as trocas de informações realizadas e ter acesso ao código YAIL.

A plataforma foi desenvolvida com o uso da biblioteca React para o *frontend*, no qual foram mapeados alguns componentes visuais do tipo *User Interface* e *Layout*. Segundo o autor foi preciso realizar uma limitação de escopo e por conta disso componentes como *Image*, *ListView*, *Notifier* não foram implementados. O projeto ficou estruturado com uma pasta englobando todos os componentes mapeados em um diretório chamado *ai_components*.

A plataforma conta também com um *backend* desenvolvido em um servidor Node.js com rotas para criar a conexão com o servidor Rendezvous, por exemplo. Além disso o *backend* também realiza a conversão do código YAIL em um objeto JSON para que facilitasse a utilização pelo *frontend*.

Segundo o autor, a maior dificuldade técnica se encontra na falta de padronização do código YAIL, complexidade e repetição do código. Por conta disso, alguns componentes não foram implementados e nem os *Procedures*, que funcionam como funções e dependem de análise do código YAIL para sua conversão e execução em JavaScript (BRANDT, 2021).

3.2 Atualização do estado da arte

Nesta seção é realizada a atualização do Mapeamento Sistemático da Literatura realizado no trabalho de Brandt (2021). É utilizado o mesmo protocolo de pesquisa adotado no trabalho anterior, com pequenas adaptações, que serão detalhadas no decorrer desta seção.

Durante o desenvolvimento da aplicação LIDEA-w, foi realizada uma pesquisa por artigos e trabalhos científicos, publicados até o ano de 2019, onde foram consideradas os seguintes conjuntos de strings para buscas:

- "android" AND "web" AND "debugging" AND ("emulator" OR "emulador") AND ("environment" OR "IDE" OR "ambiente")
- "App Inventor" AND "live debugging"
- YAIL "Young Android Intermediate Language"

Segundo o autor, o primeiro conjunto de *strings* apresentou um total de 3980 resultados. No entanto, apenas um desses resultados atendeu aos critérios de inclusão/exclusão estabelecidos por Brandt (2021). Quanto ao segundo conjunto de *strings*, ele gerou apenas cinco resultados, sendo que apenas um deles foi considerado relevante para o propósito da pesquisa. Por fim, o último conjunto resultou em seis artigos, dos quais um já havia sido incluído anteriormente.

Com base nesses resultados, o autor concluiu que não existiam artigos ou trabalhos similares à proposta em questão. Os estudos encontrados tratavam principalmente do uso do *App Inventor* em contextos diferentes do estabelecido pelo autor. Por fim, foram selecionados os seguintes trabalhos:

- Ambiente Web integrado com *App Inventor* para execução de aplicações Android (FRANÇA, 2019)
- *Live programming of mobile apps in App Inventor* (SCHILLER, ABELSON, et al., 2014)
- Ferramenta de Geração de Aplicativos Móveis Híbrido/*Cross-platform* Baseada em Editor de Interface (SPRANGER, 2019)

3.2.1 Protocolo de pesquisa

Conforme já apresentado, propõe-se elaborar um Mapeamento Sistemático da Literatura com objetivo de atualizar a revisão literária feita anteriormente. A pergunta central de pesquisa que este mapeamento sistemático busca responder é: Como são desenvolvidos os ambientes web de *live debugging* para *App Inventor*?

Foram consideradas como fonte de pesquisa os artigos indexados pela ferramenta digital *Google Scholar* e como forma de complementar a pesquisa foi utilizado Google e o repositório MIT *App Inventor - Related Research*.

Como forma de focar na atualização da pesquisa foi definido como critério de exclusão artigos publicados antes de 2018 e artigos com foco em tutoriais com intuito de ensinar o uso do *App Inventor*. Foram considerados artigos nos idiomas inglês e português cujo objetivo seja incluir informações sobre interpretação do código YAIL e propostas para realização de *live debugging* de aplicativos do *App Inventor*.

Com base na pergunta de pesquisa e nas informações apresentadas até o momento, foram selecionados termos e seus sinônimos relacionados à questão central. Esses termos estão indicados na Tabela 1. Com base nessa tabela, foram formulados conjuntos de strings que serão utilizados na execução da pesquisa.

Tabela 1: Termos utilizados nas buscas

TERMOS	INGLÊS
<i>App Inventor</i>	
Depuração	<i>debugging</i>
YAIL	
Ao vivo	<i>live</i>
Web	
Emulação	<i>emulator</i>

3.2.2 Execução da pesquisa

A execução da pesquisa foi dividida em grupos de *strings* e foi realizada inicialmente na ferramenta *Google Scholar* com o filtro “qualquer idioma” e com a data a partir de 2019.

Tabela 2: Primeira *string* de busca utilizada na pesquisa

<i>String de Busca</i>	"app Inventor" AND ("debug" OR "depuração") AND "web"
-------------------------------	---

Com a *string* de busca da Tabela 2, foram encontrados 274 resultados no total em outubro de 2022. Entretanto, apenas um trabalho relevante foi encontrado (FRANÇA, 2019) - o mesmo já foi utilizado na pesquisa realizada por Brandt (2021).

Então, adaptando o protocolo de pesquisa do trabalho anterior, uma nova busca foi realizada. Foi incluída a palavra-chave “yail” no lugar de “web,” a fim de encontrar trabalhos relacionados à linguagem utilizada pelo *App Inventor* e deixar a busca mais refinada.

Tabela 3: Segunda *string* de busca utilizada na pesquisa

<i>String de Busca</i>	"yail" AND "app inventor" AND ("debug" OR "depuração")
-------------------------------	--

Ao utilizar a *string* da tabela 3, a busca ficou tão específica que foi encontrado apenas um resultado em outubro de 2022. No entanto, o único trabalho encontrado não é relevante para pesquisa, uma vez que aborda de maneira geral conceitos do *App Inventor* e apenas cita a linguagem YAIL, sem entrar em maiores detalhes.

Com base nisso, foi realizada uma nova busca, utilizando a *string* da Tabela 4, com o objetivo de encontrar trabalhos que abordassem novas formas de depuração simultaneamente à utilização da plataforma do *App Inventor*. Nessa nova tentativa, o termo "ao vivo" ou "live" foi incluído.

Tabela 4: Terceira *string* de busca utilizada na pesquisa

<i>String de Busca</i>	"app inventor" AND ("debug" OR "depuração") AND ("ao vivo" OR "live")
-------------------------------	---

Com a utilização da *string* da Tabela 4, foram encontrados um total de 92 resultados em outubro de 2022. Novamente, foi identificado o trabalho de (FRANÇA, 2019), já citado por Brandt (2021), que se mostrou relevante para o estudo. No entanto, os demais trabalhos

encontrados basicamente tratavam da explicação do *App Inventor* e da sua utilização na construção de aplicativos, não sendo de grande utilidade para o propósito deste trabalho.

Por fim, foi realizada a última pesquisa no *Google Scholar*, utilizando agora a *string* de busca da tabela 5, a qual contém exclusivamente o termo Yail e o significado de suas siglas, a fim de encontrar potenciais trabalhos focados sobre a linguagem.

Tabela 5: Quarta e última *string* de busca utilizada na pesquisa

<i>String</i> de Busca	YAIL " <i>Young Android Intermediate Language</i> "
-------------------------------	---

Com a *string* de busca do Grupo 5, foi encontrado apenas 1 resultado em outubro de 2022. O trabalho encontrado, apenas cita superficialmente o Yail, como sendo um código, um conjunto de abstrações para Kawa. Informações já citadas no trabalho de Brandt (2021).

Tendo em vista que o resultado dessas pesquisas não apresentou nenhuma novidade em relação ao já encontrado por Brandt, foi realizada uma nova busca por trabalhos relacionados ao tema, agora na plataforma do *MIT App Inventor*, na parte de pesquisas relacionadas, ou seja, *MIT Related Research* (MIT APP INVENTOR, 2022). Esse repositório foi incluído pois engloba os trabalhos de pesquisa realizados pelo grupo de pesquisas que desenvolve o *App Inventor*.

Essa página de pesquisa é relativamente simples, exibindo todos os trabalhos de pesquisa relacionados ao *App Inventor* em uma única página, separadas por tópicos do ano de publicação da pesquisa. Nesse repositório encontrou-se os seguintes trabalhos relevantes (CLARKE e ABELSON, 2018) e (WANG, 2018). Estes foram incluídos mesmo entrando em conflito com o critério de exclusão, pois a ferramenta de pesquisa não foi utilizada no trabalho de Brandt e também não foi utilizada dentro das referências.

3.2.3 Análise do resultados

Após os resultados encontrados nas buscas de pesquisas, julgou-se relevante para utilização neste trabalho, os seguintes artigos:

- *Creating a Database Log History Page for Cloud Components in App Inventor to Support Software Development Education* (WANG, 2018)
- *Democratizing Cross-Platform Mobile Development with MIT App Inventor* (CLARKE e ABELSON, 2018)

- Plataforma Web para execução de aplicativos Android desenvolvidos com App Inventor (BRANDT, 2021)

3.2.3.1 Creating a Database Log History Page for Cloud Components in App Inventor to Support Software Development Education

Nesse artigo escrito por WANG (2018) é explicado como foi feita a criação de um ambiente web para visualização de *log* de dados para CloudDB, o qual é utilizado como componente de banco de dados em nuvem dentro do *App Inventor*.

O componente CloudDB do *App Inventor* é construído sobre o Redis - um banco de dados focado em alto desempenho, tendo como principal característica a sua agilidade para acessar e armazenar informações. Para que a plataforma web de visualização de logs fosse possível, fez-se necessário a utilização de um servidor proxy, pois assim, forneceria segurança e a abstração necessária para a interação do cliente (plataforma *web*) com o Redis.

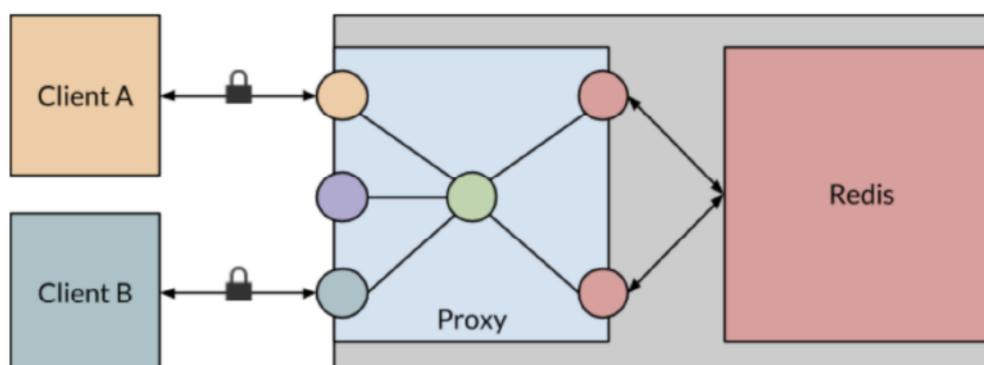


Figura 8: Interação *App Inventor* com Redis (WANG, 2018)

A interface web foi criada usando linguagens típicas de desenvolvimento *frontend*, como HTML, CSS e Javascript. Todas interações com o Redis foram feitas por um *framework* chamado Flask desenvolvido em Python, o que permitiu a interação do Redis e sites construídos em HTML e que utilizam servidor em Node.js.

O processo de comunicação da plataforma criada pelo autor se deu por meio de uso de *tokens* de autenticação do *App Inventor* e a forma como esse *log* de dados chega para ser acessada na plataforma web é via uma lista JSON.

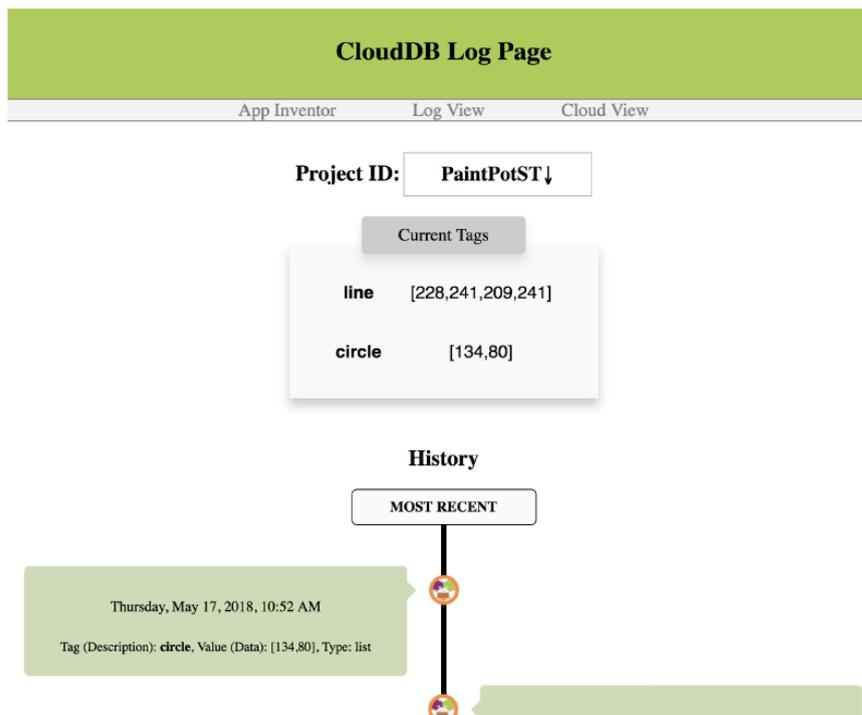


Figura 9: Plataforma *Web* Criada para visualização de logs da CloudDB (WANG, 2018)

Apesar do trabalho de WANG (2018) conter objetivos diferentes do que o deste, tem em comum o fato de ambos desenvolverem um ambiente web e utilizarem *tokens* de autenticação do *App Inventor* - o que traz um conteúdo relevante e que pode auxiliar o desenvolvimento deste trabalho.

3.2.3.2 Democratizing Cross-Platform Mobile Development with MIT App Inventor

O artigo desenvolvido por ABELSON E CLARKE (2018) traz uma forma de democratizar ainda mais o uso do *App Inventor* e o desenvolvimento de aplicativos. Essa democratização proposta pelos autores seria alcançada ao inserir a possibilidade de desenvolvimento via *App Inventor* para sistemas iOS e não só para sistema Android.

O objetivo dos autores ao desenvolver a versão para iOS é garantir a consistência e funcionalidade em ambas as plataformas (Android/iOS). Como a versão Android foi desenvolvida primeiro, os padrões do *App Inventor* são baseados no desenvolvimento dos componentes do Android. Isso implica que todas as medidas e unidades são baseadas na versão do Android, que nem sempre estão alinhadas com a versão iOS. Esse é o desafio: garantir a compatibilidade entre as duas plataformas, para que o aplicativo funcione corretamente em ambas.

O objetivo desse projeto de ABELSON E CLARKE (2018), apesar de distinto do deste trabalho, contém semelhanças, como o fato de buscar democratizar ainda mais o acesso

ao *App Inventor* e basear-se nos componentes da aplicação original desenvolvida pelo *App Inventor* para alcançar uma componentização desejada para outra plataforma/sistema e que reproduza o mesmo resultado da original.

3.3 Considerações Finais

Neste capítulo, foi apresentada a atualização da análise do estado da arte relacionada à pergunta central de pesquisa: "Como são desenvolvidos os ambientes web de *live debugging* para *App Inventor*?". Para isso, baseamo-nos no trabalho anterior realizado por Brandt (2021) por meio de um Mapeamento Sistemático de Literatura.

Embora os resultados obtidos nas pesquisas realizadas não tenham atendido plenamente aos objetivos deste trabalho, devido à escassez de conteúdo publicado e à especificidade do tema, é importante destacar que esses resultados contribuíram de forma significativa para o entendimento do *App Inventor*.

Apesar da pouca quantidade de conteúdo disponível, foi possível identificar uma certa similaridade entre os trabalhos analisados, especialmente em relação aos conceitos aplicados para a comunicação com o *App Inventor* e às metodologias descritas pelos autores em seus projetos. Essa similaridade foi relevante, pois trouxe insights valiosos e embasamento teórico para o desenvolvimento deste trabalho.

Através da análise do estado da arte, foi possível compreender como outros pesquisadores abordaram a criação de ambientes web de *live debugging* para o *App Inventor*, e isso serviu como base para o desenvolvimento do presente trabalho. Apesar das limitações encontradas, essa análise contribuiu para direcionar os esforços e fornecer referências importantes na construção de um sistema web funcional capaz de executar telas de aplicativos móveis desenvolvidos no *App Inventor*.

4 Análise e projeto

Neste capítulo é apresentado o levantamento e análise dos requisitos, seguido da estimativa, planejamento e desenvolvimento da solução técnica.

4.1 Levantamento de análise de requisitos

O levantamento de requisitos é considerado o início para toda a atividade de desenvolvimento de *software*, sendo considerado um papel importante na construção de um sistema de informação (MENDONÇA, 2014). Para realizar a análise e levantamento dos requisitos desta plataforma foram realizados análise do código já desenvolvido por Brandt (2021) e testes exploratórios nos componentes existentes na aplicação do *App Inventor*, já que segundo Sanjugtha (2020) o teste exploratório consiste em examinar todas as camadas do aplicativo enquanto pensa em todas as possibilidades de casos de uso. Cada um dos componentes visuais e não-visuais do ambiente de desenvolvimento do *App Inventor* foi explorado. A partir desses testes exploratórios, foram identificados os requisitos funcionais, para que possam ser desenvolvidos na evolução da ferramenta LIDEA-w proposta neste trabalho.

Assim, na próxima seção, são apresentados os requisitos funcionais na forma de histórias de usuário (DAVIDSON, 2014). Para cada história de usuário, são coletados os métodos e eventos de cada componente e, para os componentes visuais, uma interface de usuário utilizando o componente no *MIT Companion* é apresentada.

4.1.2 Requisitos funcionais

4.1.2.1 Componentes de interface do usuario

Nas tabelas 6 a 8 são apresentados os requisitos funcionais referentes aos componentes de *user interface* do *App Inventor* a serem implementados na plataforma LIDEA-W.

Tabela 6: Requisito funcional 1

Identificador	RF1	Título	Desenvolver o componente <i>ListView</i>
História do Usuário	Como usuário eu quero adicionar uma lista de conteúdos, texto ou imagens		
Interface	Métodos e Eventos		

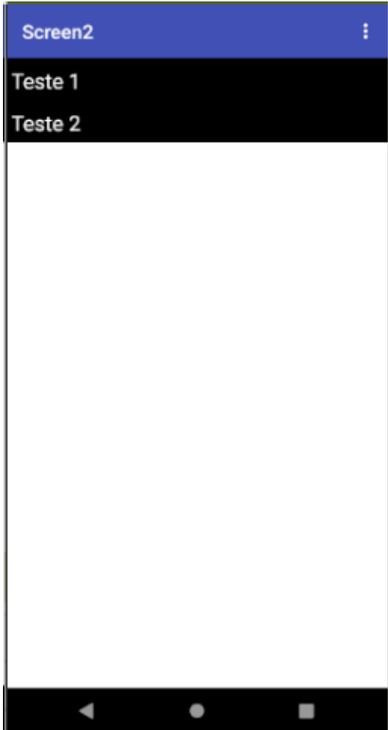
	<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>GetDetailText</i>: Retorna o detalhe de texto do elemento <i>ListView</i>; ● <i>CreateElement</i>: Cria o elemento <i>ListView</i>; ● <i>GetImageName</i>: Retorna o nome do arquivo da imagem do elemento <i>ListView</i>; ● <i>GetMainText</i>: Retorna o texto principal do elemento <i>ListView</i>; ● <i>Refresh</i>: Atualiza o conteúdo do <i>ListView</i>; <p>Eventos:</p> <ul style="list-style-type: none"> ● <i>AfterPicking</i>: Evento a ser gerado após a escolha de um elemento na lista;
---	--

Tabela 7: Requisito funcional 2

Identificador	RF2	Título	Desenvolver o component <i>Notifier</i>
História do Usuário		Como usuário eu quero receber mensagens de alerta	
Interface		Métodos e Eventos	
		<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>DismissProgressDialog</i>: Fecha o alerta criado por <i>ShowProgressDialog</i>; ● <i>ShowAlert</i>: Cria uma notificação temporária; ● <i>ShowChooseDialog</i>: Mostra uma notificação com dois botões; ● <i>ShowMessageDialog</i>: Mostra uma notificação com um botão para fechar a notificação; ● <i>ShowProgressDialog</i>: Mostra uma notificação com um título e mensagem opcional; ● <i>ShowTextDialog</i>: Mostra uma notificação com uma caixa de input; <p>Eventos:</p> <ul style="list-style-type: none"> ● <i>AfterChoosing</i>: Evento a ser gerado após a escolha do método <i>ShowChooseDialog</i>; ● <i>AfterTextInput</i>: Evento a ser gerado após a escolha do método <i>ShowTextDialog</i>; ● <i>ChoosingCanceled</i>: Evento a ser gerado após cancelar uma opção; ● <i>TextInputCanceled</i>: Evento a ser gerado após 	

	usuário cancelar métodos do <i>ShowPasswordDialog</i> ou <i>ShowTextDialog</i> ;
--	--

Tabela 8: Requisito funcional 3

Identificador	RF3	Título	Desenvolver o componente <i>WebView</i>
História do Usuário		Como usuário eu quero poder visualizar um site externo a aplicação.	
Interface		Métodos e Eventos	
 <p>Medicina e Psicologia são os cursos mais concorridos; veja a relação candidato/vaga do</p>		<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>CanGoBack</i>: Retorna true se o componente <i>WebView</i> puder voltar no histórico; ● <i>CanGoForward</i>: Retorna true se o componente <i>WebView</i> puder avançar no histórico; ● <i>ClearCaches</i>: Limpa o cache interno utilizado pelo componente; ● <i>ClearCookies</i>: Limpa os cookies utilizados pelo componente; ● <i>ClearLocations</i>: Limpa as permissões de localização armazenadas; ● <i>GoBack</i>: Volta para a página anterior da lista do histórico; ● <i>GoForward</i>: Avança para a próxima página da lista do histórico; ● <i>GoHome</i>: Carrega a página configurada nas configurações do componente; ● <i>GoToUrl</i>: O componente redireciona para o url informado; ● <i>Reload</i>: Recarrega a página; ● <i>StopLoading</i>: Parar de carregar a página; <p>Eventos:</p> <ul style="list-style-type: none"> ● <i>BeforePageLoad</i>: Evento a ser gerado e executado antes da página carregar; ● <i>ErrorOccurred</i>: Evento a ser gerado quando acontece algum erro; ● <i>PageLoaded</i>: Evento a ser gerado quando a página é carregada; ● <i>WebViewStringChange</i>: Evento a ser gerado quando o método <i>setWebViewString</i> é executado a partir do JavaScript; 	

4.1.2.2 Componentes de mídia

Nas tabelas 9 a 15 são apresentados os requisitos funcionais referentes aos componentes de mídia do *App Inventor* a serem implementados na plataforma LIDEA-W.

Tabela 9: Requisito funcional 4

Identificador	RF4	Título	Desenvolver o componente <i>Camcorder</i>
História do Usuário		Como usuário eu quero poder usar a câmera para gravar vídeos	
Interface		Métodos e Eventos	
		<p>Métodos:</p> <ul style="list-style-type: none"> • <i>RecordVideo</i>: Grava um vídeo e aciona o evento <i>AfterRecording</i>; <p>Eventos:</p> <ul style="list-style-type: none"> • <i>AfterRecording</i>: Evento gerado que indica que um vídeo foi gravado com a câmera e fornece o caminho para o vídeo; 	

Tabela 10: Requisito funcional 5

Identificador	RF5	Título	Desenvolver o componente <i>Camera</i>
História do Usuário		Como usuário eu quero poder tirar uma foto utilizando a câmera do dispositivo.	
Interface		Métodos e Eventos	

	<p>Métodos:</p> <ul style="list-style-type: none"> • <i>TakePicture</i>: Tira uma foto e aciona o evento <i>AfterPicture</i>; <p>Eventos:</p> <ul style="list-style-type: none"> • <i>AfterPicture</i>: Evento gerado após tirar a foto e fornecer o caminho para a foto;
---	---

Tabela 11: Requisito funcional 6

Identificador	RF6	Título	Desenvolver o componente <i>ImagePicker</i>
História do Usuário		Como usuário eu quero poder selecionar uma foto da minha galeria de imagens	
Interface		Métodos e Eventos	
		<p>Métodos:</p> <ul style="list-style-type: none"> • <i>Open</i>: Abre o <i>ImagePicker</i>, como se o usuário tivesse clicado nele; <p>Eventos:</p> <ul style="list-style-type: none"> • <i>AfterPicking</i>: Evento a ser gerado após a atividade <i>ImagePicker</i> retornar seu resultado e as propriedades terem sido preenchidas; • <i>BeforePicking</i>: Evento a ser gerado quando o <i>ImagePicker</i> é clicado ou o seletor é mostrado usando o método <i>Open</i>; • <i>GotFocus</i>: Indica que o cursor foi movido sobre o <i>ImagePicker</i> para que agora seja possível clicar nele; • <i>LostFocus</i>: Indica que o cursor foi movido para longe do <i>ImagePicker</i>, então não é mais possível clicar nele; • <i>TouchDown</i>: Indica que o <i>ImagePicker</i> foi pressionado; • <i>TouchUp</i>: Indica que o <i>ImagePicker</i> foi liberado; 	

Tabela 12: Requisito funcional 7

Identificador	RF7	Título	Desenvolver o componente <i>Image</i>
História do Usuário		Como usuário eu quero poder visualizar arquivos de imagem	
Interface		Métodos e Eventos	
		Eventos: <ul style="list-style-type: none"> ● <i>Click</i>: Evento que ocorre quando uma imagem é clicada 	

Tabela 13: Requisito funcional 8

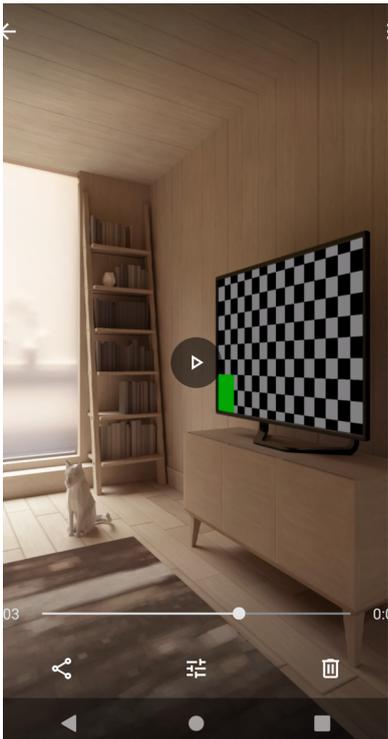
Identificador	RF8	Título	Desenvolver o componente <i>VideoPlayer</i>
História do Usuário		Como usuário eu quero visualizar vídeos existentes no dispositivo.	
Interface		Métodos e Eventos	
		Métodos: <ul style="list-style-type: none"> ● <i>GetDuration</i>: Retorna a duração do vídeo. ● <i>Pause</i>: pausa o vídeo; ● <i>SeekTo</i>: Pula o andamento do vídeo para o momento especificado; ● <i>Start</i>: Inicia o vídeo; ● <i>Stop</i>: Volta o vídeo ao início e pausa; Eventos: <ul style="list-style-type: none"> ● <i>Completed</i>: Evento gerado indica que o vídeo chegou ao final; 	

Tabela 14: Requisito funcional 9

Identificador	RF9	Título	Desenvolver o componente <i>Player</i>
História do Usuário		Como usuário eu quero poder executar um arquivo de áudio do dispositivo.	
Interface		Métodos e Eventos	

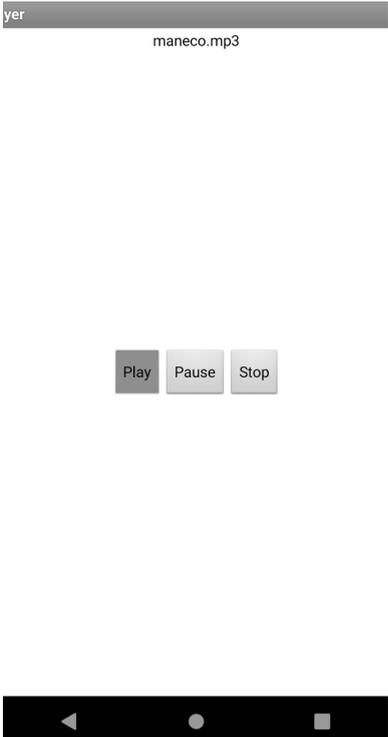
	<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>Pause</i>: Pausa a mídia que está tocando; ● <i>Start</i>: Inicia a mídia selecionada; ● <i>Stop</i>: Finaliza a mídia que está tocando; ● <i>Vibrate</i>: Vibra o dispositivo pelo tempo especificado; <p>Eventos:</p> <ul style="list-style-type: none"> ● <i>Completed</i>: Evento é gerado quando a mídia é executada até o final. ● <i>OtherPlayerStarted</i>: Evento é gerado quando outro player de mídia é iniciado;
---	--

Tabela 15: Requisito funcional 10

Identificador	RF10	Título	Desenvolver o componente <i>Sound</i>
História do Usuário		Como usuário eu quero poder executar um arquivo de áudio com poucos segundos.	
Interface		Métodos e Eventos	
		<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>Pause</i>: Pausa a mídia que está tocando; ● <i>Resume</i>: Retoma a reprodução do som após uma pausa; ● <i>Start</i>: Inicia a mídia selecionada. ● <i>Stop</i>: Finaliza a mídia que está tocando; ● <i>Vibrate</i>: Vibra o dispositivo pelo tempo especificado; <p>Eventos: <i>Sem eventos</i></p>	

4.1.2.3 Componentes de armazenamento

Na tabela 16 são apresentados o requisito funcional referente ao componente de armazenamento do *App Inventor* a serem implementados na plataforma LIDEA-W.

Tabela 16: Requisito funcional 11

Identificador	RF11	Título	Desenvolver o componente <i>TinyDB</i>
História do Usuário		Como usuário eu quero poder armazenar dados usando chave e valor	
Interface		Métodos e Eventos	
Component sem interface		<p>Métodos:</p> <ul style="list-style-type: none"> ● <i>ClearAll</i>: Limpa toda base de dados; ● <i>ClearTag</i>: Limpa a entrada dado a <i>tag</i>(chave); ● <i>StoreValue</i>: Armazena valor fornecido para uma determinada <i>tag</i> (chave); ● <i>GetValue</i>: Recupere o valor armazenado sob a <i>tag</i> (chave) fornecida; <p>Eventos: <i>Sem eventos</i></p>	

4.1.2.4 Implementações na lógica de interpretação do código Yail

Nas tabelas 17 a 19 são apresentados os requisitos funcionais referentes às alterações necessárias na lógica da aplicação responsável por interpretar o código YAIL proveniente da aplicação *backend* e convertê-la em comandos JavaScript.

Tabela 17: Requisito funcional 12

Identificador	RF12	Título	Refatoração das condicionais
História do Usuário		Como usuário eu quero ter os comandos de condição dos blocos do <i>App Inventor</i> sendo interpretados dentro da aplicação	
Comando de bloco			

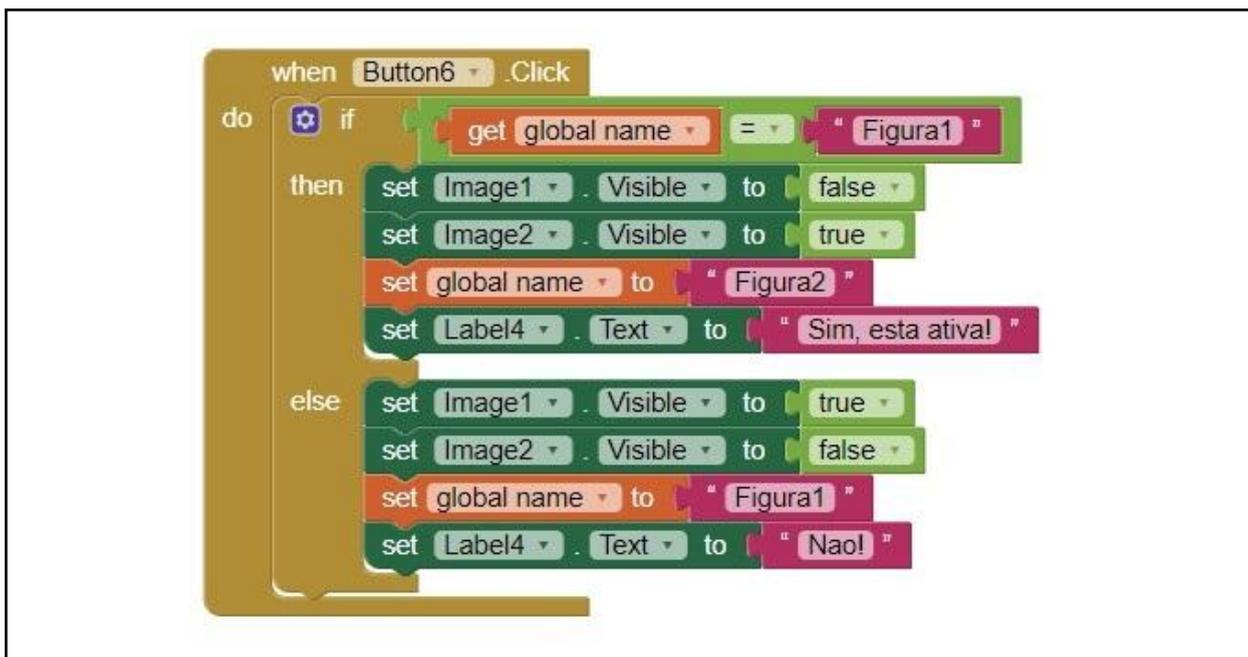
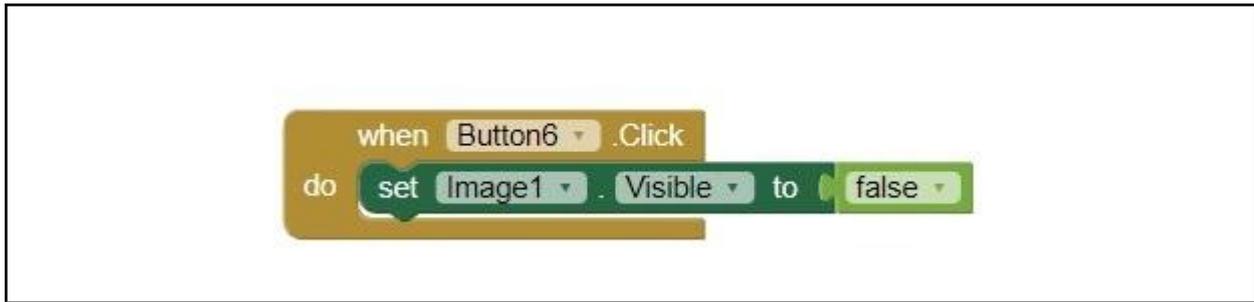


Tabela 18: Requisito funcional 13

Identificador	RF13	Título	Refatoração variáveis globais
História do Usuário	Como usuário eu quero ter as variáveis globais do <i>App Inventor</i> sendo interpretadas e alteradas dentro da aplicação		
Comando de bloco			
<pre> initialize global name to "Figura1" when Button6 .Click do set global name to "Figura2" </pre>			

Tabela 19: Requisito funcional 14

Identificador	RF14	Título	Alterar visibilidade dos componentes
História do Usuário	Como usuário eu quero poder alterar a visibilidade dos componentes a partir de ações de blocos.		
Comando de bloco			



4.2 Estimativa

Nesta seção é apresentado a estimativa do desenvolvimento dos requisitos funcionais apresentados na seção anterior. Para o gerenciamento do projeto de desenvolvimento foi escolhido um dos principais frameworks ágeis para gerência de projetos de software, o SCRUM (SCHWABER, 2020) e como ferramenta para gerenciamento foi usado a ferramenta de *Kanban* do *GitLab*.

O Scrum para Schwaber (2020) é um framework ágil para gerência de projetos que se destaca por apoiar-se em três conceitos ágeis: transparência, controle e adaptação. Existem várias etapas definidas para o funcionamento desse *framework*, mas uma das etapas é o planejamento e a estimativa que um item do *backlog* precisa. O *product backlog* é uma lista emergente e ordenada do que é necessário para melhorar o produto (SCRUM.ORG, 2019). O *backlog* é a única fonte de trabalho realizada pelo Time Scrum, e para estimar os itens de *backlog* pode-se utilizar os pontos de história. Os pontos de história (*story point*) representam o tamanho estimado para implementar uma tarefa, nesse caso um requisito funcional e também pode ser entendido como uma medida de complexidade (DAVIDSON, 2014).

Os pontos de história foram estimados de forma coletiva pelos autores deste trabalho para o planejamento do desenvolvimento dos requisitos e são definidos utilizando a sequência de Fibonacci. Na tabela 20 é exposto o valor dos pontos de história utilizados e uma breve definição.

Tabela 20: Tabela de definição dos pontos de história

Story Point	Definição
1	Tarefas simples no qual não requer quase nenhum esforço e conhecimento prévio
2	Intermediário entre 1 e 3
3	Tarefas simples, conhecimento prévio, mas requer um pouco de esforço

5	Tarefa com conhecimento prévio e requer esforço
8	Tarefa com conhecimento prévio e requer muito esforço
13	Tarefa sem conhecimento prévio, insegurança para realizar, mas possui entendimento de como iniciá-la
21	Tarefa sem conhecimento prévio e sem entendimento de como iniciá-la

Com base na estrutura de pontos de histórias apresentada na tabela 20, foi realizada a estimativa para cada requisito funcional na tabela 21.

Tabela 21: Tabela de definição dos pontos de história para cada requisito funcional

Requisito funcional	<i>Story Point</i>
RF1	8
RF2	8
RF3	8
RF4	21
RF5	21
RF6	13
RF7	21
RF8	13
RF9	8
RF10	8
RF11	21
RF12	21
RF13	13
RF14	13

Setup Inicial	✓								
RF1 - Desenvolver o componente <i>ListView</i>		✓							
RF2 - Desenvolver o component <i>Notifier</i>		✓	✓						
RF3 - Desenvolver o componente <i>WebViewer</i>			✓						
RF5 - Desenvolver o componente <i>Camera</i>			✓	✓					
RF7 - Desenvolver o componente <i>Image</i>				✓					
RF9 - Desenvolver o componente <i>VideoPlayer</i>					✓				
RF4 - Desenvolver o componente <i>Camcorder</i>					✓				
RF9 - Desenvolver o componente <i>Player</i>						✓			
RF10 - Desenvolver o componente <i>Sound</i>						✓			
RF6 - Desenvolver o componente <i>ImagePicker</i>						✓			
RF13 - Alterar visibilidade dos componentes							✓		
RF14 - Refatoração variáveis globais							✓		
RF11 - Desenvolver o componente <i>TinyDB</i>							✓	✓	
RF12 - Refatoração das condicionais								✓	✓

5.2 Desenvolvimento

Nesta seção, a implementação do projeto é apresentada, bem como as dificuldades encontradas ao longo do desenvolvimento.

5.2.1 ListView

O componente *ListView* é responsável por exibir na interface do *App Inventor* uma lista de itens, a qual pode vir acompanhada de imagens ou não. Também fornece a opção de realizar uma busca pelo nome do item e possui um método que é executado ao clicar em algum item da lista.

Para o desenvolvimento desse componente foi necessário utilizar os componentes *List*, *ListItem*, *ListAvatar* e *ListItemText* para a lista em si e o *InputLabel* e *Input* para fazer o campo de busca da lista, todos fornecidos pela biblioteca *MaterialUI* para o *React*.

```
return (
  <div className={componentClass}>
    {showFilterBar && (
      <FormControl className={classes.form}>
        <InputLabel className={classes.font}>Search list</InputLabel>
        <Input
          id="search-input"
          className={classes.font}
          onChange={handleChange}
        />
      </FormControl>
    )}
    <List id={componentName}>
      {list.map((item, index) => (
        <ListItem
          key={index + 1}
          button
          onClick={() => {
            handleClickItem(item.text, index);
          }}
        >
          {item.img && (
            <ListItemAvatar>
              <Avatar alt={item.img.name} src={item.img.url} />
            </ListItemAvatar>
          )}
          <ListItemText
            primary={
              <Typography className={classes.font}>{item.text}</Typography>
            }
            secondary={
              <Typography className={classes.fontDetail}>
                {item.detail}
              </Typography>
            }
          />
        </ListItem>
      )}
    </List>
  </div>
);
```

Figura 10: Imagem do código do componente *ListView*

A implementação da parte visual do componente foi relativamente simples, apenas foi necessário ler a documentação do *Material UI* para saber como utilizar os componentes fornecidos. O desafio ficou por conta de como realizar a ação de clicar em algum item da lista e o valor desse item ser transmitido para uma variável global.

Para isso foi necessário recorrer a um recurso do *React* chamado *Context API*, para assim conseguir alterar o estado da variável global com o valor do item clicado. A parte de buscar um item dentro da lista foi resolvida utilizando o método *filter* de arrays do próprio *Javascript*, no conteúdo do Input que estava associada a um estado do componente, com isso foi possível fornecer uma busca reativa e instantânea na tela da aplicação.

5.2.2 Notifier

No caso do componente *Notifier*, o desafio inicial estava relacionado à necessidade de acionamento por meio de uma função chamada por outro componente, o *Button*. Devido a essa dependência, os métodos descritos no requisito funcional desse componente precisam ser acessíveis fora do escopo do componente *Notifier*. A arquitetura existente do projeto não permitia o encadeamento de funções associadas a outros componentes, pois os componentes estavam isolados. Para acessá-los, eram utilizadas referências de identificadores únicos por meio do *Document Object Model (DOM)*.

Considerando que o projeto utiliza a biblioteca *Material UI* para definir a aparência visual dos componentes, foi decidido utilizar o componente chamado *Dialogs*, que consiste em um tipo de janela modal que aparece à frente do conteúdo do aplicativo para fornecer informações importantes ou solicitar uma decisão. No entanto, esse componente utiliza um conceito denominado Portal, que permite renderizar os elementos em um nó *DOM* que existe fora da hierarquia *DOM* do próprio componente. Essa abordagem tornou o componente inacessível de acordo com a arquitetura inicialmente planejada para o projeto já que dessa forma o identificador usado para o *notifier* estaria em um *DOM* diferente.

Para contornar esse problema foi utilizado *Context API* que é uma funcionalidade oferecida pelo *React*. Em vez de utilizar a propagação de propriedades através de múltiplos componentes, a *Context* cria um contexto, que é um objeto que contém os dados que serão compartilhados. Esse contexto pode ser acessado por qualquer componente que esteja dentro do escopo desse contexto.

5.2.3 WebViewer

A implementação do *WebViewer* utilizou o elemento HTML chamado *iframe*, que possibilita o carregamento de outra página HTML dentro do DOM. De forma essencial, o elemento *iframe* permite a inclusão de uma página da web dentro da página principal. A implementação do *WebViewer* foi realizada de maneira simples, sem apresentar nenhum problema significativo.

5.2.4 Camcorder

O componente *Camcorder* é responsável por fazer uso da câmera do dispositivo e realizar a gravação de vídeos dentro do *App Inventor*. Ele possui um método de gravar vídeo, chamado *RecordVideo*, que após o término de sua execução emite um evento chamado *AfterRecording* que indica que um vídeo foi gravado pela câmera e fornece o caminho da mídia recém gravada para ela poder ser exibida na interface da aplicação.

```

return (
  <camera 86 (
    <div style={{ textAlign: "center" }}>
      <Webcam
        audio={false}
        ref={webcamRef}
        videoConstraints={videoConstraints}
      />
      {capturing ? (
        <button style={{ padding: "7px" }} onClick={handleStopCaptureClick}>
          Parar
        </button>
      ) : (
        <button
          style={{ padding: "7px" }}
          onClick={handleStartCaptureClick}
        >
          Gravar
        </button>
      )}
    </div>
  )}
);

```

Figura 11: Imagem do código do componente *Camcorder*

Para execução deste componente foi necessário recorrer a uma biblioteca externa chamada *React-WebCam*, ela tem o intuito de abstrair códigos complexos para manipulação de imagens e vídeos provenientes da webcam do usuário e assim facilitar o uso da interação usuário-aplicação.

A dificuldade encontrada no desenvolvimento desse componente ficou por conta da transformação da mídia recém gravada em dados binários para poder transmitir ao componente *VideoPlayer*, que é responsável por executar mídias de vídeo. Foi necessário recorrer a documentação da biblioteca externa *React-WebCam* para saber a ordem correta de execução dos métodos fornecidos e como transformar os dados da mídia. Para conseguir transmitir os dados para o componente *VideoPlayer* foi necessário criar um *Context API* global de eventos que aconteciam na aplicação, assim os novos dados e alterações foram transmitidos por meio dele.

5.2.5 Camera

Dentro do *App Inventor* também existe outro componente que utiliza a câmera do dispositivo, ele se chama *Camera* e tem a função de tirar fotos. Possui um método de tirar foto chamado *TakePicture*, que após a sua execução emite um evento chamado *AfterPicture* que indica que uma foto foi tirada pela câmera e fornece o caminho da mídia a ser exibida na interface da aplicação.

Este componente é muito similar ao *Camcorder*, por isso, também utiliza a biblioteca *React-Webcam*, com a diferença que agora utiliza os métodos referentes à captura de imagem. Na figura abaixo é possível observar o código do componente *Camera*.

```
return (
  <div style={{ textAlign: "center" }}>
    <Webcam
      audio={false}
      ref={webCamRef}
      screenshotFormat="image/jpeg"
      videoConstraints={videoConstraints}
    />
    <button style={{ padding: "7px" }} onClick={capture}>
      Tirar Foto
    </button>
  </div>
);
```

Figura 12: Imagem do código do componente *Camera*

Para que esse componente funcione dentro do *App Inventor* ele deve trabalhar em conjunto com mais dois componentes: o *TinyDb* responsável por armazenar no *storage* (armazenamento local) do dispositivo dados binários referentes a uma mídia e com o componente *Image*, responsável por exibir na interface uma imagem. Assim como o componente *Camcorder* o grande desafio no desenvolvimento desse componente foi a comunicação com os demais, por isso, também foi necessário recorrer ao *Context API* do *React*, para conseguir avisar os outros sobre os dados compartilhados.

Dessa forma o fluxo para funcionamento desse componente foi: usuário tira foto com a webcam, o evento *AfterPicture* é emitido e responsável por fornecer ao *TinyDb* os dados binários da imagem recém tirada, este por sua vez fornece o caminho relativo da imagem para o componente *Image* que exibe na tela a foto recém tirada.

5.2.6 VideoPlayer

O *VideoPlayer* é um componente que é utilizado para reproduzir um vídeo dentro do *App Inventor*. Possui métodos responsáveis por iniciar, pausar, pular o andamento e recomeçar (volta o vídeo ao início e pausa) de um vídeo. Trabalha em conjunto com outros componentes, como é o caso do *Camcorder*, responsável por gravar um vídeo.

Esse componente foi desenvolvido utilizando o elemento `<video>` nativo do HTML, portanto não foi necessário recorrer a nenhuma biblioteca externa. Na figura abaixo é possível observar a estrutura do componente, também é possível notar que ele funciona de duas maneiras, uma quando o componente *VideoPlayer* é invocado passando a url do vídeo e outra que trabalha em conjunto com o vídeo vindo da *webcam*, ou seja, o vídeo gravado pelo *Camcorder*.

```
return (
  <
    {videoPath?.url} && (
      <div>
        <video
          id="videoControl"
          controls
          src={videoPath?.url}
          width={width}
          height={height}
          style={{ display: visible }}
          muted
        ></video>
      </div>
    )
    {videoFromWebCam} && (
      <div>
        <video
          id="videoControl"
          controls
          src={videoFromWebCam}
          width="100%"
          height="100%"
          style={{ display: visible }}
          muted
        ></video>
      </div>
    )
  </
);
```

Figura 13: Imagem do código do componente *VideoPlayer*

Quando o componente recebe vídeo vindo da webcam ele utiliza os dados compartilhados pelo *ContextAPI*, já quando o vídeo vem configurado direto pelo *App Inventor*, ele já recebe a url de forma direta, não é necessário recorrer ao contexto.

O maior desafio para esse componente foi como transmitir as ações de manipulação de vídeo, ou seja, os métodos (iniciar, pausar, recomeçar). Logo foi necessário atribuir um *id* chamado *videoControl* ao elemento do HTML, para posteriormente conseguir manipular suas ações via *Javascript* por meio de alterações via dom, como podemos observar na figura 14.

```

// Video Player
if (commands.includes("VideoPlayer")) {
  const methodValue = commands.split(" ")[2].split("(")[0];
  const video = window.document.getElementById("videoControl");
  let action = "";
  if (video) {
    if (methodValue === "Start") {
      action = `document.getElementById("videoControl").play()`;
    }
    if (methodValue === "Pause") {
      action = `document.getElementById("videoControl").pause()`;
    }
    if (methodValue === "Stop") {
      action = `document.getElementById("videoControl").pause();
      document.getElementById("videoControl").currentTime = 0`;
    }
  }
  return `${action}`;
}

```

Figura 14: Transmissão dos métodos do componente *VideoPlayer*

5.2.7 Image

Para a implementação do componente de imagem, foi necessário um estudo e uma investigação mais aprofundados no funcionamento do *App inventor* e do *AI2 Companion*, já que era preciso entender como era feito o envio dos arquivos entre essas duas aplicações. Para isso foi realizada uma engenharia reversa.

Usando a guia de rede dentro do navegador *web* foi identificada a presença de um método HTTP específico, denominado PUT, que desempenha um papel fundamental no contexto do envio de arquivos binários. O método PUT é uma operação do protocolo HTTP que permite a criação ou substituição de recursos em um servidor, por meio do envio de dados. No âmbito do App Inventor, esse requisição PUT é utilizado para enviar as imagens inseridas pelos usuários para o servidor. Essas imagens são tratadas como "asset" no contexto do App Inventor, representando recursos essenciais para o pleno funcionamento do aplicativo.

Durante a análise detalhada do código-fonte do App Inventor, foi possível identificar a implementação específica dessa requisição PUT e compreender como as imagens eram manipuladas nesse contexto. A compreensão minuciosa dos mecanismos e protocolos envolvidos no processo de envio de arquivos foi de suma importância para assegurar a correta implementação do componente de imagem no escopo do projeto.

Importante ressaltar que o fragmento de código apresentado na Figura 15, obtido por meio da análise do código-fonte do App Inventor, desempenhou um papel fundamental ao fornecer insights sobre o funcionamento da requisição PUT e sua interação com o envio e tratamento das imagens. Essa investigação permitiu uma visão mais precisa dos aspectos técnicos envolvidos, norteando as etapas subsequentes da implementação.

```

var conn = goog.net.XmlHttp();
var arraybuf = new ArrayBuffer(blob.length);
var arrayview = new Uint8Array(arraybuf);
for (var i = 0; i < blob.length; i++) {
  arrayview[i] = blob[i];
}
var rs = top.ReplState;
var encoder = new goog.Uri.QueryData();
//var z = filename.split('/'); // Remove any directory components
//encoder.add('filename', z[z.length-1]); // remove directory structure
var z = filename.slice(filename.indexOf('/') + 1, filename.length); // remove the asset di
encoder.add('filename', z); // keep directory structure

if (rs.proxy) {
  rs.proxy.postMessage(['asset', encoder.toString(), arraybuf], rs.proxy_origin);
  success(); // What happens if we fail?
} else {
  var conn = goog.net.XmlHttp();
  conn.retries = 3;
  conn.open('PUT', rs.baseurl + '?' + encoder.toString(), true);
  conn.onreadystatechange = function () {
    if (this.readyState == 4 && this.status == 200) {
      if (success) { // process callbacks
        success();
      }
    } else if (this.readyState == 4) {
      if (this.retries > 0) {
        this.retries--;
        this.open('PUT', rs.baseurl + '?' + encoder.toString(), true);
        this.send(arraybuf);
      }
    }
  }
}

```

Figura 15: Trecho do código fonte do método putAsset do App Inventor

Com base na análise realizada, foi implementada uma rota no backend do projeto para permitir que o servidor envie os arquivos para o backend da aplicação LIDEA-W. Essa rota é responsável por tratar as requisições PUT enviadas pelo App Inventor e processar os arquivos de imagem recebidos.

```

router.put('/', async (req, res, next) => {
  req.on('data', async (data) => {
    const pathImg = path.resolve(
      __dirname,
      '..',
      'uploads',
      req.query.filename
    );
    fs.appendFileSync(pathImg, Buffer.from(data), 'binary');
  });

  medias.add(req.query.filename);

  res.send({
    status: 'MEDIA',
  });
});

```

Figura 16: Trecho do código fonte do método put da plataforma LIDEA-W

O código apresentado na Figura 16 é responsável por tratar a requisição PUT enviada para o servidor LIDEA-W. Ao receber essa requisição, o código executa uma série de instruções para processar e armazenar a imagem enviada. Dentro dessa função, o código obtém o caminho do arquivo de imagem onde ele será armazenado. Esse caminho é determinado com base no nome do arquivo, que é obtido a partir do parâmetro filename presente na query da requisição.

Em seguida, o código utiliza a biblioteca fs para gravar os dados da imagem no arquivo especificado pelo caminho pathImg. A função fs.appendFileSync é usada para adicionar os dados da imagem ao arquivo, sendo que Buffer.from(data) converte os dados recebidos em formato binário.

Dessa forma, foi possível armazenar as imagens, assim como outros tipos de mídia que estavam anexadas no projeto dentro do App Inventor no backend. Entretanto, vale salientar que essa requisição responsável por enviar os dados binários das mídias, só é executada pelo App Inventor apenas uma vez, no momento em que se conecta ao AI Companion, ou seja, caso novas mídias forem adicionadas ao projeto, será necessário desconectar e conectar novamente para que o upload seja realizado.

Os uploads foram armazenados em uma rota estática dentro do backend, o que permite que eles sejam acessados externamente por outras aplicações, ou ainda ao digitar o caminho relativo da mídia desejada no navegador. Com isso um arquivo de imagem, chamado imagem1.png que está na pasta uploads dentro do backend, pode ser acessado ao se digitar no navegador: <http://enderecobackend/uploads/imagen1.png> e foi justamente esse endereço http o enviado ao frontend, junto com as informações do código YAIL que já eram enviadas

anteriormente. Assim, o componente imagem, que até então tinha apenas uma imagem fixa padrão, pode ser refatorado, recebendo valores dinâmicos.

5.2.8 Player

O *App Inventor* possui dois componentes responsáveis por reproduzir áudios, um deles é o *Player*, o qual é indicado para áudios maiores como é o caso de músicas. O *Player* é um componente não visível, por isso não possui nenhum tipo de interface gráfica na aplicação. Os botões de pausar, iniciar, recomeçar, que geralmente um *player* possui, devem ser inseridos pelo usuário, que também deve atribuir na ação de *click* do botão a chamada para os métodos respectivos a cada função de manipulação do áudio.

Foi utilizado o elemento `<audio>` nativo do HTML, portanto, não foi necessário recorrer a nenhuma biblioteca externa. Para deixar esse componente o mais fiel ao original do *App Inventor* foi necessário mudar a visibilidade padrão do elemento `<audio>` para *hidden*, assim nada seria exibido na interface, entretanto, ainda seria possível escutar o áudio na aplicação por meio da manipulação do dom, via *Javascript*.

```
return (  
  <  
    <audio  
      controls  
      id={componentName}  
      src={soundPath?.url}  
      style={{ visibility: "hidden" }}  
    ></audio>  
  </>  
);  
}
```

Figura 17: Imagem do código do componente *Player*

O maior ponto de atenção no desenvolvimento desse componente foi como transmitir ao elemento HTML os métodos acionados pelo usuário. Para isso utilizou-se uma estratégia muito parecida à utilizada no *VideoPlayer*, conforme podemos ver na imagem abaixo.

```

// Player
if (commands.includes("Player")) {
  const regexComponent = /call-component-method\s+'([^']*)/;
  const matchComponent = commands.match(regexComponent);
  const componentName = matchComponent[1].trim();

  const methodValue = commands.split(`call-component-method `)[1];
  const regex = /'([^']*).*\(\s*/;
  const match = methodValue.match(regex);
  const methodName = match[1].trim();

  const sound = window.document.getElementById(componentName);
  let action = "";
  if (sound) {
    if (methodName === "Start") {
      action = `document.getElementById("${componentName}").play()`;
    }
    if (methodName === "Pause") {
      action = `document.getElementById("${componentName}").pause()`;
    }
    if (methodName === "Stop") {
      action = `document.getElementById("${componentName}").pause();
      document.getElementById("${componentName}").currentTime = 0`;
    }
  }
  return `${action}`;
}

```

Figura 18: Transmissão dos métodos do componente *Player*

Os comandos do componente são executados via manipulação do dom, executando funções nativas do elemento `<audio>`.

5.2.9 Sound

O outro componente responsável por reproduzir áudio dentro do *App Inventor* é o *Sound*. Ao contrário do *Player* o *Sound* é indicado para áudios curtos. Ele também é um componente não visível, por isso não possui nenhum tipo de interface gráfica na aplicação. Seu funcionamento a estrutura do componente é exatamente a mesma do *Player*.

A diferença entre os dois componentes é que enquanto o *Player* possui os métodos: *Pause*, *Start*, *Stop* o *Sound* possui *Pause*, *Resume* e *Start*. Na prática usando esse componente não podemos notar diferença entre eles, os desafios para o desenvolvimento desse componente foram os mesmos do anterior.

5.2.10 Image Picker

O componente *ImagePicker* é um elemento especializado que atua como um botão com uma finalidade específica dentro de um sistema. Ao ser acionado pelo usuário, o *ImagePicker* invoca a exibição da galeria de imagens do dispositivo, oferecendo ao usuário a oportunidade de selecionar uma imagem de sua escolha. Após a seleção da imagem, esta é armazenada e a propriedade "*Selection*" assume o nome do arquivo no qual a imagem foi armazenada

Quanto à implementação em si, a complexidade foi relativamente baixa. O componente utiliza o recurso de botão disponibilizado pela biblioteca de interface *Material UI*, combinado com o elemento HTML `<input>` do tipo "file". O trecho de código abaixo ilustra essa implementação:

```
const handleClick = useCallback(() => {
  inputRef.current.click();
}, []);

return (
  <div>
    <input
      type="file"
      id="file"
      ref={inputRef}
      style={{ display: "none" }}
      accept="image/*"
      onChange={handleImage}
    />
    <Button
      onClick={handleClick}
      id={componentName}
      variant="contained"
      className={classes.button}
    >
      {properties.text}
    </Button>
  </div>
);
```

Figura 19: Trecho de código do componente *Image Picker*

Dessa forma, ao clicar no botão "Selecionar Imagem", o comportamento padrão do elemento HTML `<input>` do tipo "*file*" é acionado, permitindo a escolha da imagem desejada

pelo usuário, proporcionando uma funcionalidade simples, mas eficiente, para a escolha e armazenamento de imagens em um aplicativo.

O desafio enfrentado nesse componente consistiu em atualizar o estado das mídias de forma a adicionar a imagem selecionada e torná-la acessível para o componente de imagem. Como mencionado anteriormente, os componentes possuem funções e propriedades isoladas. A solução encontrada foi criar uma função dentro do Contexto de Conteúdo (*ContentContext*) responsável por atualizar o array de mídias.

```
const updateMedias = (mediaItem, componentName) => {
  setMedia((prevMedia) => [...prevMedia, mediaItem]);
  const componentToUpdate = components.find(
    (item) => item.componentName === componentName
  );
  if (!componentToUpdate) {
    return;
  }

  const updatedProperties = [
    ...componentToUpdate.componentProperties.filter(
      (item) => item.propertyName !== "Picture"
    ),
    {
      propertyName: "Picture",
      propertyValue: mediaItem.name,
      propertyType: "text",
    },
  ];

  setComponents((prevComponents) => [
    ...prevComponents.filter((item) => item.componentName !== componentName),
    {
      ...componentToUpdate,
      componentProperties: updatedProperties,
    },
  ]);
};
```

Figura 20: Trecho de código da função *updateMedias*

Essa função, *updateMedias* (figura 20), é responsável por adicionar a *mediaItem* ao array de mídias através da função *setMedia*. Em seguida, localiza-se o componente a ser atualizado dentro do *array* de *components*, utilizando o nome do componente (*componentName*). As propriedades do componente são atualizadas, removendo a propriedade *"Picture"* (imagem anterior) e adicionando a nova imagem selecionada. Por fim,

o estado de components é atualizado com as propriedades atualizadas do componente correspondente.

Essa abordagem permitiu a atualização eficiente do estado das mídias e das propriedades do componente de imagem, solucionando o desafio de forma adequada.

5.2.11 *TinyDb*

Para a implementação do *TinyDB*, utilizou-se o *IndexedDB*, um recurso nativo dos navegadores, em conjunto com a biblioteca IDB. O *IndexedDB* é uma API poderosa para o armazenamento de dados no navegador, enquanto a biblioteca IDB facilita a interação com essa API, simplificando o processo de manipulação dos dados.

```
const dataBase = await openDB(dbName, dbVersion, {
  upgrade(db) {
    if (!db.objectStoreNames.contains(storeName)) {
      const store = db.createObjectStore(storeName, {
        keyPath: "tag",
        autoIncrement: true,
      });
      store.createIndex("tag", "tag", { unique: true });
    }
  },
});

setDb(dataBase);
```

Figura 21: Trecho de código da inicialização do *indexedDB*

Essa abordagem possibilitou o desenvolvimento de um mecanismo eficiente e confiável de armazenamento persistente por meio do *TinyDB*. Ao utilizar o *IndexedDB* e a biblioteca IDB, criou-se um ambiente adequado para a retenção dos valores das variáveis entre as execuções do aplicativo, proporcionando uma experiência contínua e consistente aos usuários.

Adicionalmente, é relevante mencionar que os itens de dados armazenados no *TinyDB* consistem em *tags* e valores. Para armazenar um item de dados, é necessário especificar a *tag* sob a qual ele deve ser armazenado. Essa *tag* é um bloco de texto que fornece um nome identificador aos dados. Posteriormente, é possível recuperar os dados que foram armazenados sob uma determinada *tag*, utilizando a mesma *tag* como referência. Essa

abordagem possibilita um acesso eficiente e organizado aos dados armazenados, facilitando a recuperação de informações específicas quando necessário.

Para evitar problemas de acesso às funções existentes nesse componente, foi utilizada a abordagem já mencionada de *Context*. Dessa forma as funções passam a estar disponíveis para todos os componentes que necessitem realizar armazenamento de informações (Figura 22).

```
return (
  <TinyDbContext.Provider value={ { storeValue, getValueTag, cleanTag } }>
    {children}
  </TinyDbContext.Provider>
);
```

Figura 22: Trecho de código do *Context* do *TinyDB*

5.2.12 Refatoração das condicionais

Uma etapa muito importante e desafiadora no desenvolvimento deste trabalho foi algumas mudanças e melhorias na lógica de interpretação do código *Yail*. Essas mudanças possibilitaram adicionar funcionalidades e melhorar o funcionamento do LIDEA-W, a primeira alteração foi a refatoração das condicionais.

```
(init-runtime)(define-event Button1 Click()(set-this-form)
  (if (call-yail-primitive yail-equal? (*list-for-runtime* (get-property 'TextBox1
'Text) "")) '(any any) "=") (begin (set-and-coerce-property! 'Label1 'Text "01\u00e1,
estranho!" 'text)) (begin (set-and-coerce-property! 'Label1 'Text (call-yail-primitive
string-append (*list-for-runtime* "01\u00e1, " (get-property 'TextBox1 'Text) ) '(text
text) "join") 'text))))(call-Initialize-of-components 'Screen1 'VA1 'TextBox1 'Label1
'Button1 'VerticalArrangement1)
```

Figura 23: Exemplo de sintaxe de condicionais no YAIL

Na figura acima podemos ver um exemplo de como é a sintaxe de uma condicional dentro do YAIL, essa é a forma mais básica de uma condição possível, um *if (x =y) executa algo*. Entretanto, a complexidade da sintaxe de interpretação das condicionais dentro do YAIL aumenta à medida que adicionamos métodos, ações dentro dos blocos condicionais, resultando em uma difícil tarefa na hora de traduzir o código para *Javascript*. Para realizar a tradução do código YAIL na aplicação, o LIDEA-W foi desenvolvido trabalhando com dicionários de palavras chaves, como podemos ver na figura abaixo.

```

const localVariableIndicator = "(let ( ";
const setPropertyIndicator = "(set-and-coerce-property! ' ";
const setGlobalVariableValue = "(set-var! g$";
const conditionalIndicator = "(if (call-yail-primitive ";
const haveAConditional = "(if #";
const listIndicator = "call-yail-primitive yail-list";
const SPLIT_CHARACTER = "character";
const SPLIT_SPACES = "spaces";
const SPLIT_FIRST = "first";
const EQUAL = "yail-equal";
const NOT_EQUAL = "yail-not-equal";
const LESS_THAN = "<";
const LESS_EQUAL = "≤";
const GREATER_THAN = ">";
const GREATER_EQUAL = "≥";
const BITWISE_AND = "and";
const BITWISE_OR = "or";
const BITWISE_XOR = "xor";
const callComponentMethod = "(call-component-method";
const runTimeTextProperties = "(*list-for-runtime*";

```

Figura 24: Exemplo de algumas palavras-chave no dicionário para traduzir o YAIL

É possível observar que existem algumas constantes que se refere à condicionais como o *conditionalIndicator* e *haveAConditional*, elas são utilizadas em métodos específicos responsáveis pela tradução. Foi justamente nesses métodos específicos que foram refatorados e melhorados, a fim de conseguir traduzir condições mais complexas.

Na imagem abaixo, podemos observar uma das alterações realizadas no código original de interpretação das condicionais, foi adicionado uma verificação para saber se dentro do bloco da condição está sendo chamado algum método, dentro dessa verificação existem outras verificações adicionadas como verificar se dentro desse bloco acontece alguma ação que resulta na troca de variável global, muda alguma propriedade de algum componente. Uma das estratégias para melhorar a interpretação dessas condicionais foi recorrer ao uso de *Regex*, para identificar padrões e assim melhorar a capacidade de tradução do código.

```

// Possui condicional >, <, =, etc
else if (commands.startsWith(conditionalIndicator, i)) {
  parsedCommand = getConditionalCommands(commands, variables, i);

  if (parsedCommand.includes(callComponentMethod)) {
    const regex = /\{[^}]*\}/g;
    const matches = parsedCommand.match(regex);

    let condition = parsedCommand.split(" ")[1];
    let instruction = ["", ""]; // [0] representa if; [1] representa else

    matches.forEach((match, index) => {
      let execute = match;
      if (match.includes(callComponentMethod)) {
        execute = parsedCallComponentMethod(match);
        instruction[index] += `${execute};`;
      }
      if (match.includes(setGlobalVariableValue)) {
        execute = setGlobalVariable(match, variables);
        instruction[index] += `${execute};`;
      }
      if (match.includes(setPropertyIndicator)) {
        const property = match.split("set-and-coerce-property!")[1];
        const propertyFix = `set-and-coerce-property!${property}`;
        execute = getPropertyCommands(propertyFix, variables, i - 1);
        instruction[index] += `${execute};`;
      }
    });

    const final = `if ${condition} ${instruction[0]}else${instruction[1]}`;
    parsedCommand = final; // uma string como todo comando da condicional a ser exec
  }

  parsedCommand = parsedCommand.replaceAll("{", "{");
  parsedCommand = parsedCommand.replaceAll("}", "}");
  parsedCommand = parsedCommand.replaceAll(";;", ";");

  if (parsedCommand.includes("updateProperties"))
    parsedCommand = parsedCommand.replaceAll("' '", "'");

  if (parsedCommand.includes("updateGlobalVar"))
    parsedCommand = parsedCommand.replaceAll(`""`, `"`);

  stack.push(parsedCommand);

```

Figura 25: Alguma das melhorias na tradução de condicionais do YAIL

Com essa e outras alterações similares foi possível expandir a capacidade de interpretação das condicionais dentro da aplicação, assim resultando em uma maior forma de usar blocos condicionais no *App Inventor* e que fossem traduzidas dentro do LIDEA-W.

5.2.13 Alterar visibilidade dos componentes

É muito comum em qualquer aplicação que envolva uma interface, poder a partir de uma ação fazer com que um componente apareça ou desapareça. Para conseguir inserir essa feature no projeto foi necessário estudar como as propriedades estavam sendo armazenadas no Context API padrão da aplicação.

Foi observado que por padrão os componentes quando montados dentro da interface do *App Inventor*, eram montados no LIDEA-W sem nenhum atributo que indicasse que ele estava visível, entretanto, quando o contrário era feito, a aplicação recebia uma propriedade que indicava que o componente estava não visível. Então, o primeiro passo foi corrigir esse detalhe, para daí prosseguir para algum meio de alterar a visibilidade do componente.

Para isso foi necessário implementar um método chamado *updateProperties*, com três parâmetros: nome do componente, propriedade a ser alterada e o novo valor. Esse método foi implementado dentro do arquivo de contexto de estados gerais da aplicação, na imagem abaixo é possível o código.

```
const updateProperties = (componentName, property, newValue) => {
  const update = [...components];
  update.forEach((component) => {
    if (component.componentName === componentName) {
      component.componentProperties.forEach((prop) => {
        if (prop.propertyName === property) {
          prop.propertyValue = newValue;
        }
      });
    }
  });
  setComponents(update);
};
```

Figura 26: Método responsável por alterar a visibilidade dos componentes

Esse método podia ser invocado dentro do próprio componente ou ainda dentro da lógica responsável por interpretar o código YAIL, dessa maneira foi possível alterar de forma descomplicada o estado do componente. Na figura abaixo é possível o método sendo invocado para dentro da função chamada *getPropertyCommands*, responsável por interpretar os dados dos componentes.

```
case "Visible":
  const visibility = commands
    .substring(i)
    .split("Visible ")[1]
    .substring(0, 2);

  setProperty = `updateProperties('${componentAction}', 'Visible', '${visibility}')`;
  if (commands.includes("#t (begin)" || commands.includes("#f (begin)"))
    setProperty = "";
  break;
default:
  break;
```

Figura 27: Método *updateProperties* sendo emitido

5.2.14 Refatoração variáveis globais

Uma demanda muito importante em uma aplicação é a utilização de variáveis, pois é necessário armazenar valores das operações realizadas, para por exemplo conseguir exibir informações na interface, além de toda sua importância na construção de um algoritmo de um programa. Por essa razão, foi necessário colocar essa nova *feature* como uma das prioridades dentro do LIDEA-W.

Dessa forma o passo inicial foi inserir no dicionário de interpretação de YAIL um novo valor, para conseguir detectar quando a string recebida tivesse uma variável global e então pudesse ser identificado o seu valor inicial. Logo após essa etapa, foi necessário criar um método similar ao de mudar a visibilidade dos componentes, mas dessa vez o propósito seria alterar o valor da variável global, conforme a ação executada na interface pelo usuário.

O maior desafio dessa implementação foi ajustar o algoritmo para identificar a mudança da variável dentro de um bloco condicional, pois como já abordado anteriormente, a sintaxe das condições dentro do código do YAIL é muito peculiar e muda conforme o que está dentro dos blocos. Essa alteração contribuiu de forma significativa para que os componentes pudessem interagir melhor entre si, principalmente, quando usado em condicionais para alterar valores de um componente ou ainda exibir informações na tela.

5.3 Testes Funcionais

Para avaliar os componentes desenvolvidos e as alterações na interpretação do código a partir dos requisitos funcionais, foi necessário levar em consideração a dificuldade encontrada de interação entre os componentes devido a arquitetura inicial em que o projeto foi desenvolvido. Com isso, foram realizados testes funcionais dos componentes de maneira isolada e, quando possível, testes de interação foram realizados de maneira com que se comunicassem com os outros componentes desenvolvidos ou com as alterações realizadas na parte lógica de interpretação do código YAIL deste trabalho.

Assim, para a realização dos testes dos componentes foi desenvolvida uma aplicação-modelo no App Inventor, contendo os componentes implementados e foi testada a sua funcionalidade no LIDEA-W. O projeto dessa aplicação modelo em formato .aia está disponível para download em: https://drive.google.com/file/d/1xsPq6EexNn4vTJ8_Y0IVbeEPFDrhsmiy/view?usp=drive_1ink. Por sua vez o código-fonte da aplicação desenvolvida está disponível em: <https://codigos.ufsc.br/gqs/lidea-w>

5.3.1 Testes de Integração por meio de uma Aplicação-modelo

Conforme já apresentado, para realizar os testes de integração dos componentes desenvolvidos, para os quais isso era possível, foi desenvolvida uma aplicação-modelo no App Inventor e foi avaliado o resultado da execução dessa aplicação no LIDEA-W. Os seguintes componentes foram testados utilizando a aplicação-modelo: *ListView*, *Image*, *Sound*, *Player*, *VideoPlayer* e suas funções básicas, e também foi possível testar as alterações executadas na lógica de interpretação das condições.

A aplicação-modelo desenvolvida simula os primeiros passos de um possível curso introdutório de ensino de lógica e de pensamento computacional para alunos do ensino fundamental. Nesse curso, poderia ser utilizada a ferramenta *App Inventor* para introduzir os alunos aos primeiros passos no mundo do desenvolvimento de software.

A ideia dessa aplicação é utilizar de um menu básico onde, ao clicar, seria alterado o conteúdo para as principais fontes de mídia que existem nos dias atuais em computadores e dispositivos móveis: imagens, vídeos e áudio. Além de inserir para os alunos conceitos de programação como: variáveis, métodos e condições.

A figura 28 mostra a aplicação-modelo na tela inicial do *App Inventor*:

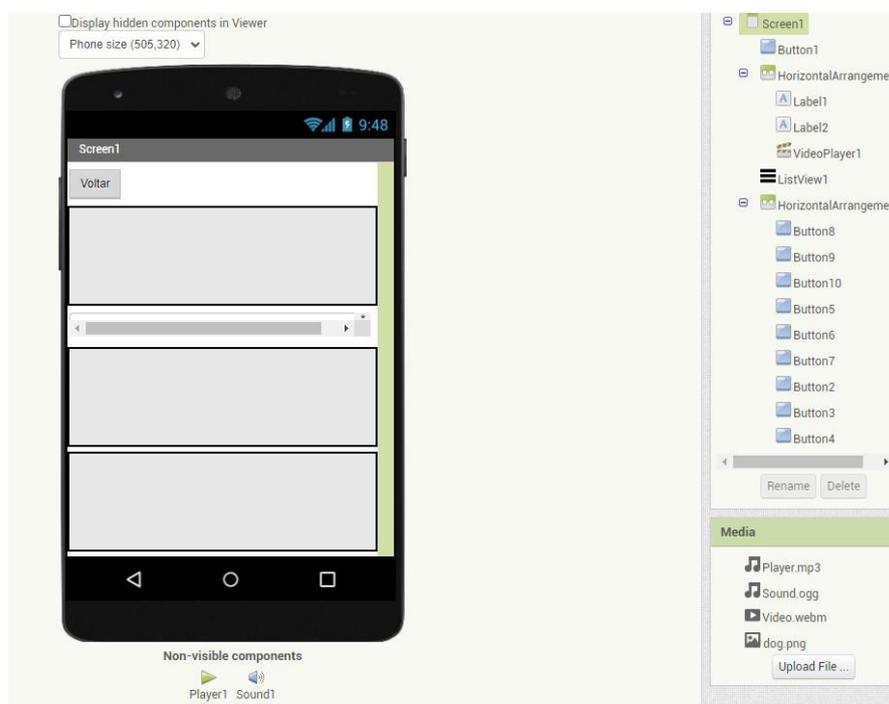


Figura 28: Imagem do aplicativo no *App Inventor*

Para o funcionamento da aplicação proposta, foram gerados onze blocos de ações e a inicialização de uma variável global. Na imagem abaixo é possível ver todos blocos criados para que essa aplicação funcionasse, serão detalhados posteriormente.

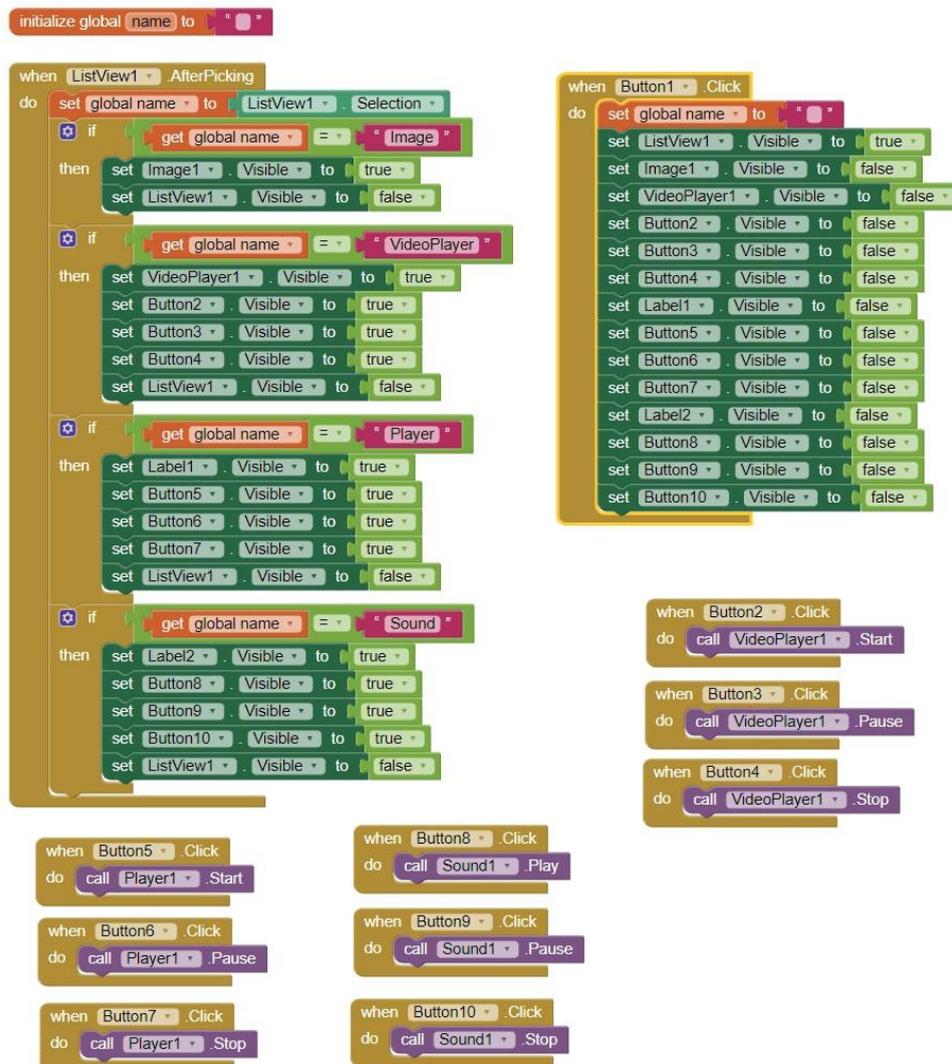


Figura 29: Imagem dos blocos do aplicativo no *App Inventor*

Nesses blocos da figura 29 estão os comandos de condição, emissão dos métodos, inicialização de variável global e sua alteração. A figura 30 mostra o resultado da tela inicial do *app* visualizado no LIDEA-W.

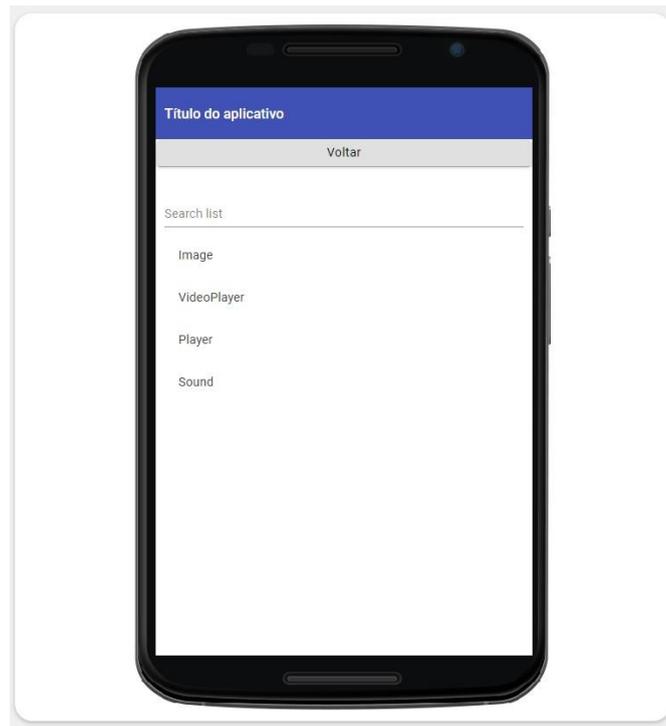


Figura 30: Imagem do aplicativo na aplicação LIDEA-W

A tela inicial exibe apenas componente *ListView* (RF -1), no qual é exibida uma lista que é configurável pela interface do *App Inventor*; que está configurada para mostrar a barra de busca, na qual o usuário consegue procurar por um valor e exibir os resultados dessa lista. Na figura 31 é possível observar a configuração da lista no *App Inventor*.

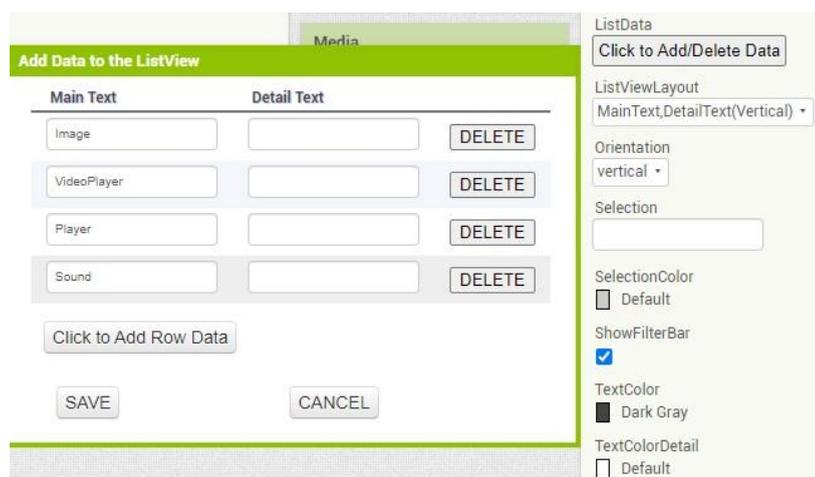


Figura 31: Imagem do ListView no *App Inventor*

A figura 32, exibe a primeira etapa necessária na parte dos blocos, que foi inicializar a variável global, com o nome “*name*” e com valor inicial uma *string* vazia.



Figura 32: Imagem da inicializacao da variavel global



Figura 33: Imagem da inicialização do *ListView1*

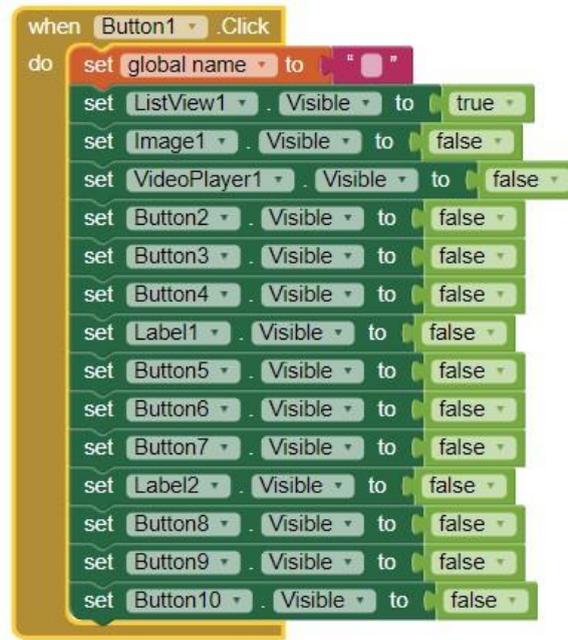


Figura 34: Imagem da inicialização do *Button1*

A figura 33 exibe o bloco de comandos referente ao *ListView* e ao *Button* rotulado “Voltar”, que tem o papel de voltar a tela inicial que exibe a lista. A lista, ao ter um dos seus itens clicados na interface, executa o método “*AfterPicking*” que transmite justamente o item selecionado. O bloco realiza os comandos: altera o valor da variável global para o valor clicado na lista, depois checa a condição, caso o novo valor da variável global seja igual ao que está em algum bloco de condição, os comandos do bloco são executados. Dessa maneira, se o item na lista clicado for o “*Image*”, ele altera o valor da variável global para “*Image*”. entrará no bloco do `if get global name = “Image”` e mudará a visibilidade da *Image1* para “*true*”. e a visibilidade da *ListView* para “*false*”, com isso teremos a tela representada pela imagem 35.

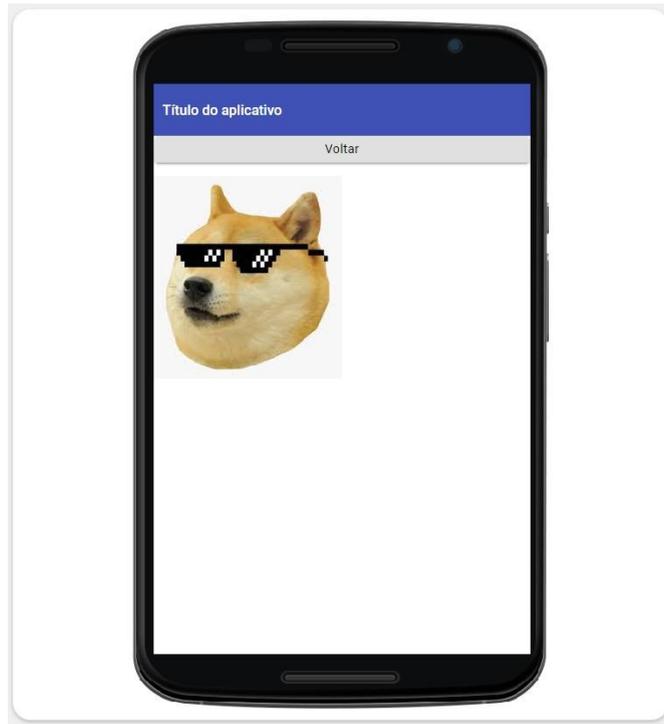


Figura 35: Imagem da aplicação

Para retornar à tela inicial com a *ListView*, é necessário clicar no botão “Voltar” exibido no topo da aplicação, com isso o bloco representado na figura 34, é executado. Ele muda o valor da variável global para uma *string* vazia, resetando-a para o valor inicial e também muda a visibilidade de todos os componentes possíveis na aplicação para “*false*”, com exceção da visibilidade da *ListView* que volta a ser “*true*”, para que possa aparecer na tela.

Quando se clica no elemento da lista *Player*, ele funciona idêntico a forma descrita anteriormente para exibição da imagem, e então exibe os botões de ação de um player, que são o “*Play*”, “*Pause*” e “*Stop*”. O componente *Player* é não visível, então por sua vez esses botões foram adicionados com componentes *Buttons* no *App Inventor*, ou seja, não fazem parte do componente e logo fica a escolha do usuário colocá-los ou não.

Para configurar o som que será tocado ao executar a função *.Start* do *Player* é necessário configurar na interface, conforme a figura 36.

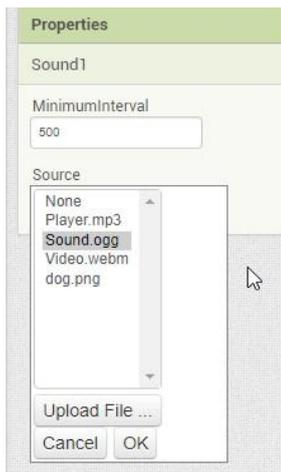


Figura 36: Imagem das propriedades do *Player*

É possível observar a tela que exibe os botões responsáveis por executar os métodos do componente *Sound* na figura 37.

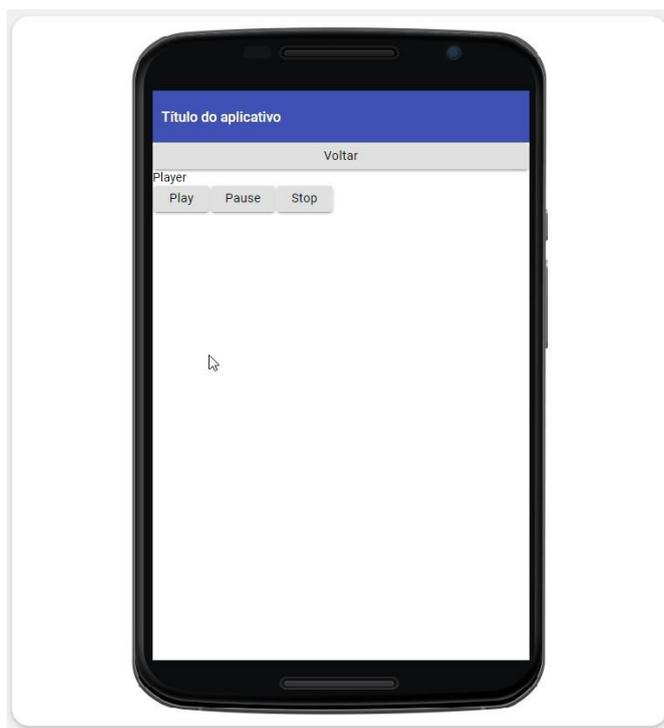


Figura 37: Imagem do componente *Sound*

Ao clicar em algum dos botões da tela, ele emite o método a ser executado do *Player*, como por exemplo, se o botão *Pause* for clicado ele emitirá o método *.Pause* do componente e assim ele vai parar o conteúdo da mídia de áudio que está sendo executado.



Figura 38: Imagem dos botões de controle do *Player*

Na Figura 38, pode-se observar as telas do *VideoPlayer* e *Sound*. Elas possuem a mesma dinâmica das telas anteriores explicadas e seus blocos de comandos seguem a mesma lógica.

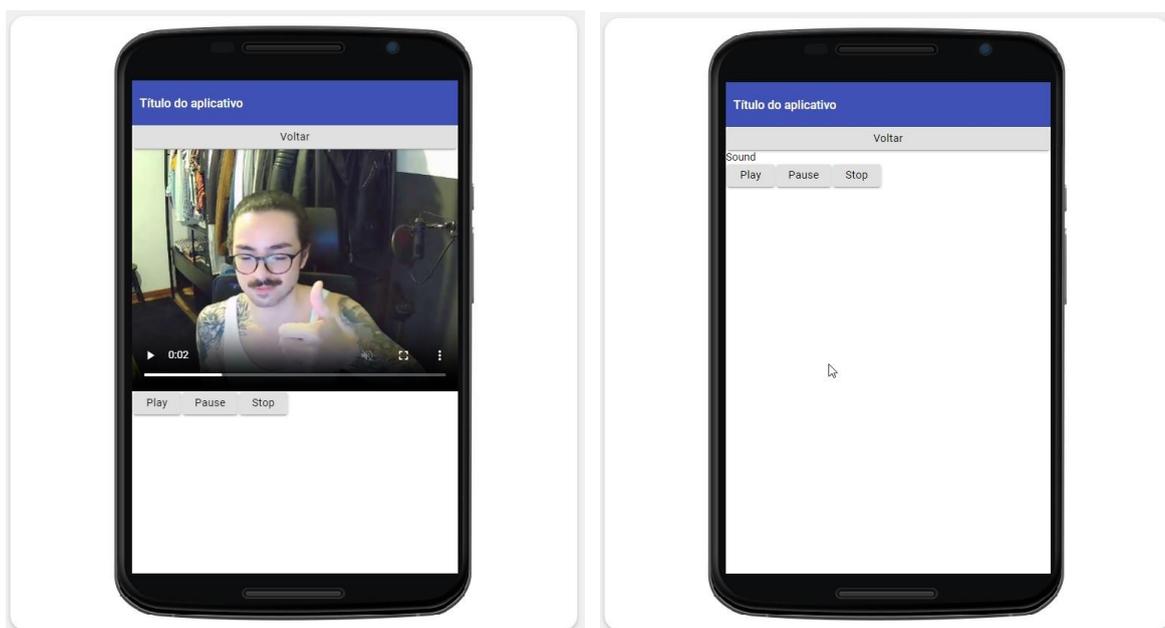


Figura 38: Imagem das as telas do *VideoPlayer* e *Sound*

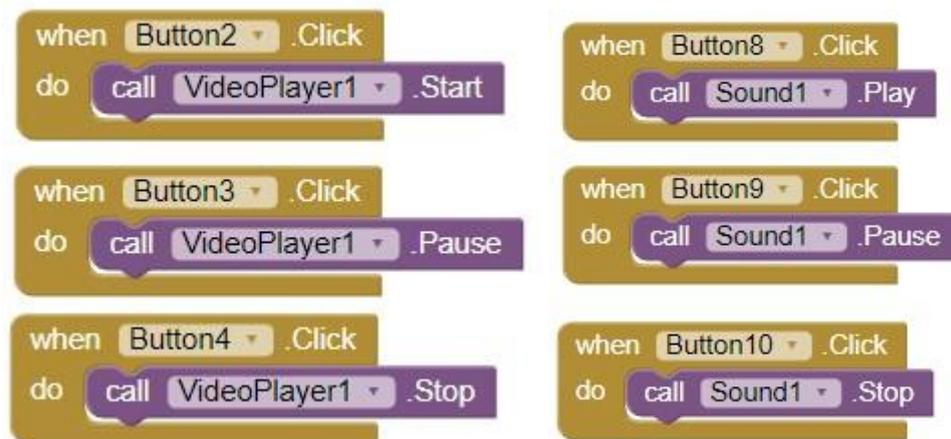


Figura 39: Imagem dos blocos das funções básicas do *VideoPlayer* e *Sound*

Com a implementação desta aplicação-modelo foi possível testar os componentes: *ListView*, *Image*, *Sound*, *Player*, *VideoPlayer* e suas funções básicas, e também foi possível testar as alterações executadas na lógica de interpretação das condições.

Na criação dos blocos no *App Inventor*, o bloco de quando a lista é clicada poderia ter sido feita de maneira diferente. Ao invés de repetir em todos blocos de condições o comando para alterar a visibilidade da lista para *false*, seria melhor se esse comando estivesse sido executado fora dos blocos, logo após a checagem de todas as condições, entretanto, isso não foi possível devido a uma limitação do algoritmo de interpretação do YAIL, que não foi capaz de identificar esse padrão e acabava não executando a ação.

O projeto dessa aplicação modelo em formato .aia pode ser baixado em: https://drive.google.com/file/d/1xsPq6EexNn4vTJ8_Y0IVbeEPFDrhsmiy/view?usp=drive_link

5.3.2 Testes unitários

Os demais componentes desenvolvidos neste trabalho, foram testados utilizando uma abordagem de testes unitários manuais, de maneira isolada, ou seja, não foram testados dentro de um contexto em que interagem com outros componentes totalmente diferentes dos responsáveis pela sua ativação e funcionamento. Isso foi necessário devido a dificuldade encontrada de interação entre os componentes, por causa da arquitetura em que o projeto foi desenvolvida e da forma em que o código YAIL foi interpretado na aplicação, o que acabou limitando as interações. Com isso, optou-se por realizar os testes funcionais dos demais componentes de maneira isolada. Os seguintes componentes foram testados de maneira

isolada: Notifier, Webviewer, Image Picker, TinyDb, Camera, Camcorder e outras condicionais.

O arquivo de testes desses componentes de maneira isolada está disponível para download em formato .aia em:

<https://drive.google.com/file/d/1j06q_BgiD7ObEARr-zatFthnDB0Olz26/view?usp=drive_link>. Os testes unitários manuais de cada componente são apresentados a seguir.

5.3.2.1 Notifier

O teste unitário do *Notifier* envolveu a implementação de dois botões com a função de clique e a ativação de duas funções disponíveis no *Notifier*, representadas em dois blocos, como ilustrado na figura 40.

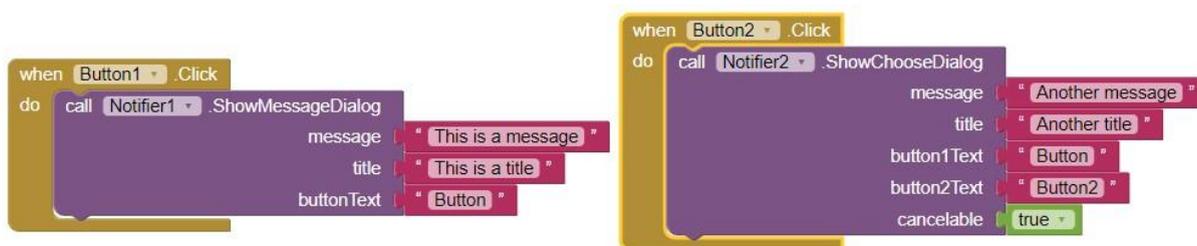


Figura 40: Imagem dos blocos usados no teste do componente *Notifier*

Como resultado, obteve-se a seguinte renderização dos *Notifiers* apresentados nas imagens da Figura 41 na ferramenta Lidea-w:

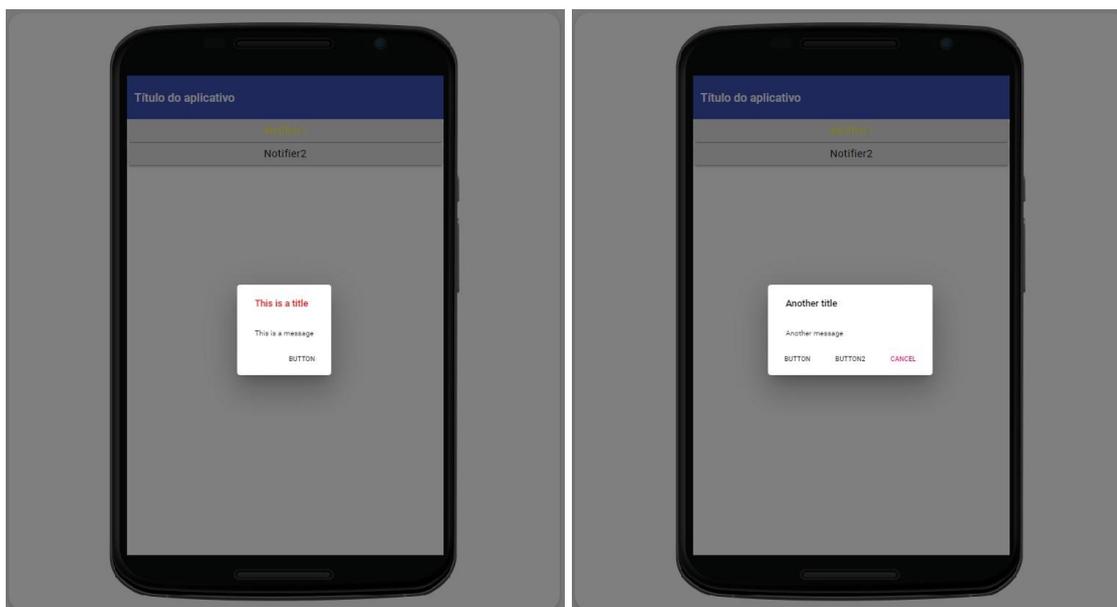


Figura 41: Imagens do resultado do componente *Notifier* no LIDEA-W

Apesar do sucesso do teste unitário, constatou-se que ao empregar o componente com inicializações distintas da função de clique do botão, ocorre uma falha na interpretação da chamada da função. Essa limitação torna o componente incompleto para ser utilizado em aplicações de maior complexidade.

5.3.2.2 Web Viewer

O componente de *Web Viewer* foi submetido a um teste unitário manual devido à sua aplicação altamente específica. Durante o teste, foi observado que o componente enfrenta problemas somente quando ocorre o bloqueio da URL utilizada para a incorporação, devido a questões de segurança. Isso significa que o componente não conseguirá exibir corretamente o conteúdo da URL bloqueada, resultando em limitações na sua funcionalidade.

No entanto, é importante ressaltar que, quando a URL utilizada é permitida para a incorporação, o componente apresenta resultados satisfatórios, conforme evidenciado na figura 42.

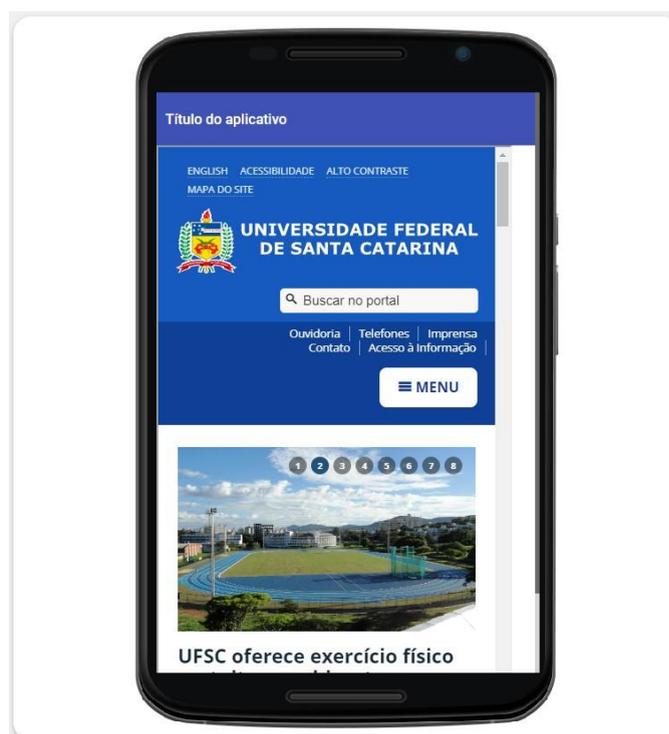


Figura 42: Resultado do *Webviewer*

5.3.2.3 Image Picker & TinyDb

Quanto aos componentes *Image Picker* e *TinyDb*, foram validados de forma isolada e não apresentaram problemas. Seus funcionamentos individuais atenderam às expectativas. No

entanto, ao testá-los em conjunto em uma aplicação mais completa, constatou-se uma falha na persistência dos dados quando ocorreu uma persistência anteriormente.

Essa falha significa que, quando o *Image Picker* e o *TinyDb* são utilizados em conjunto e já houve uma persistência anterior de dados, ocorre um problema na manutenção dessas informações. Isso resulta em uma perda de consistência e integridade dos dados armazenados.

5.3.2.4 Camera

O componente *Camera* foi avaliado através de um teste unitário manual pois se comportou de forma inesperada quando utilizado em uma tela na qual deveria aparecer ou desaparecer, conforme interação do usuário, que foi o caso do primeiro tópico abordado neste capítulo. O componente apresentava duplicidade da imagem fornecida pela webcam caso a tela também possuísse o componente *Camcorder*, mesmo este estando sem visibilidade.

Sendo assim optou-se por fazer o teste isolado do componente, no qual ele teve seu funcionamento esperado dentro do normal, sem qualquer tipo de problema detectado. Na figura 43 é possível observar os blocos do componente.

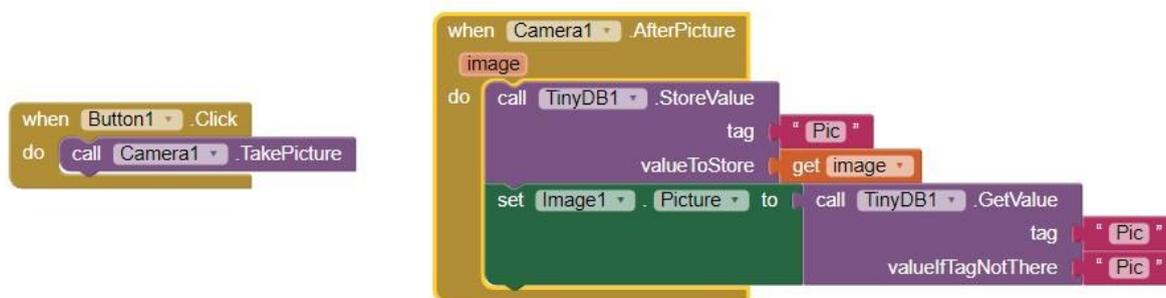


Figura 43: Blocos do componente *Camera*

O componente é acionado quando o botão 1 é clicado, ele ativa o método *TakePicture*, responsável por abrir a câmera do dispositivo, ou seja, a webcam - conforme podemos ver na imagem abaixo. A imagem da webcam tem as proporções da tela de modo proporcionar uma melhor experiência de uso e se assemelhar ao uso de um *smartphone*, ao clicar no botão “Tirar Foto” a imagem é exibida na tela da aplicação. Se o usuário desejar pode clicar novamente no botão “Abrir webcam” e tirar outra foto.

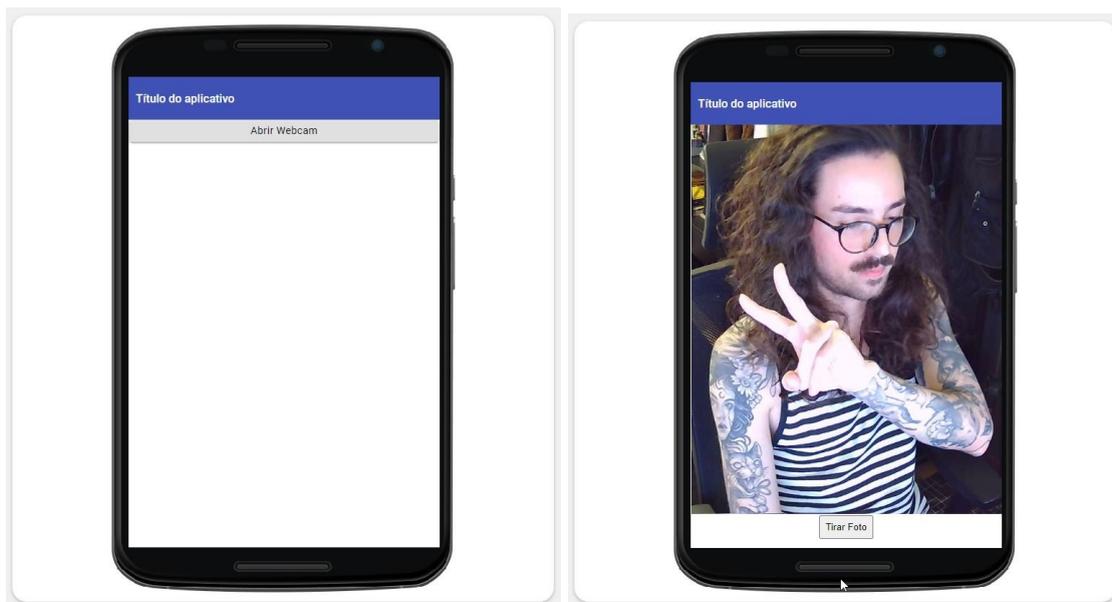


Figura 44: Resultado do componente de *Camera* na aplicação LIDEA-W

Após a foto ser tirada ela é armazenada usando o componente *TinyDb*, o qual salva os dados binários da mídia no navegador e posteriormente esses dados são passados ao componente *Image*, o que permite que a foto recém tirada fique visível na tela após o clique.

5.3.2.5 Camcorder

O componente *Camcorder* foi submetido a um teste unitário manual pelo mesmo motivo do componente *Camera*, e se comportou de maneira inesperada quando deveria aparecer ou desaparecer conforme interação do usuário, a tela de *live preview* gerada pela webcam não desaparecia mesmo quando o componente estava com a visibilidade falsa.

Entretanto, durante o teste, o componente se comportou como o esperado e não teve problemas algum, seu funcionamento foi bem similar a de quando visto no *smarthphone* ou emulador do *App Inventor*.



Figura 45: Blocos do componente *Camcorder*

A figura 45 representa os blocos do componente *Camcorder*, o componente começa a gravar a imagem do *live preview* da webcam quando clica no botão 1, que emite o

RecordVideo e assim a imagem começa a ser gravada. Quando o botão “Parar” ele para de gravar o vídeo e emite o *AfterRecording*, que é responsável por alterar o componente *VideoPlayer* que trabalha junto com o *Camcorder*, assim exibindo o vídeo na tela conforme as imagens da Figura 46.

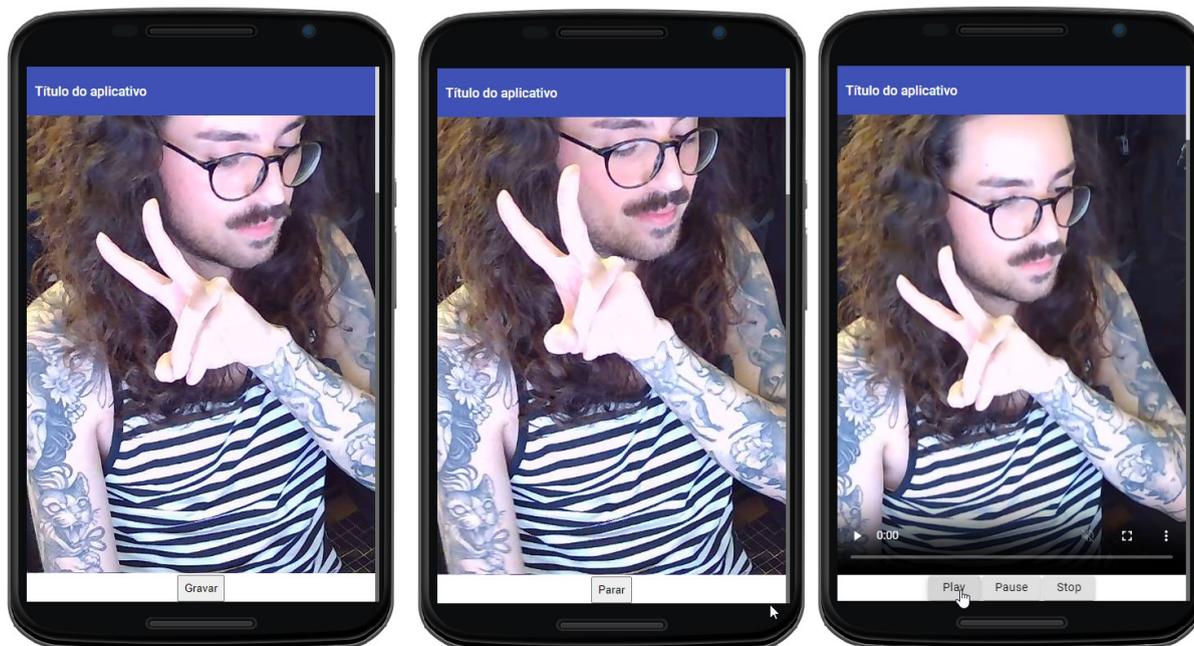


Figura 46: Exibição do Video na aplicação LIDEA-W

Os botões: *Play*, *Pause* e *Stop* estão presentes para conseguir visualizar o vídeo recém gravado e eles são colocados na tela por meio dos componentes de *Button* do *App Inventor*, ao serem clicados na interface emitem os eventos *Start*, *Pause* e *Stop*, respectivamente.

5.3.2.6 Outras condicionais

Outro teste que se fez necessário e foi realizado de maneira isolada, consistiu em avaliar a execução de comandos dentro dos blocos *if / then / else* do *App Inventor*, uma vez que o teste inicial deste capítulo abordou apenas vários blocos com *if/then*.

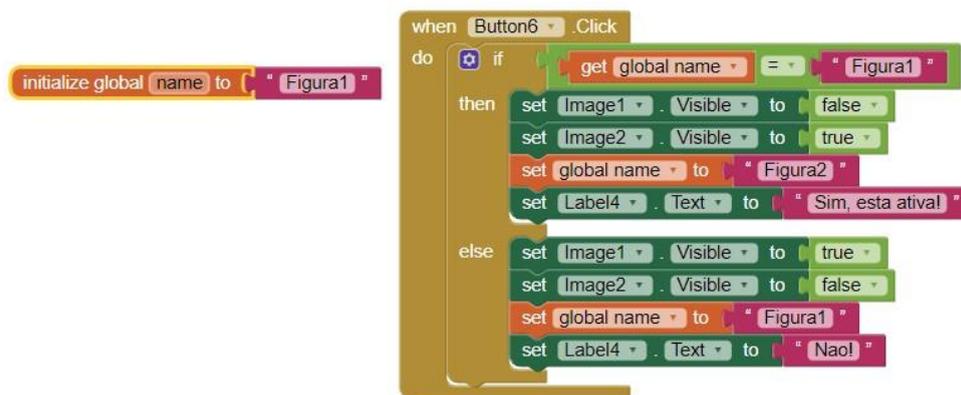


Figura 47: Blocos de IF/else no App inventor

A ideia deste teste consistia em ao clicar no botão, ele verificar qual é o valor da variável global e caso o valor seja um, exibir uma imagem, caso seja outro exibir outra imagem. Além de exibir um valor diferente na *Label* da aplicação.

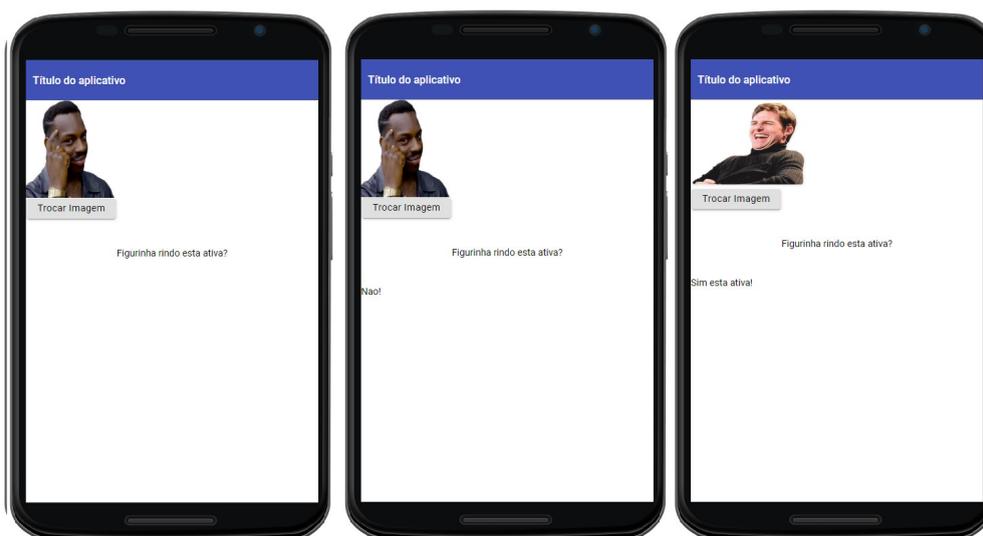


Figura 48: Resultado dos blocos de IF/else no LIDEA-W

Portanto, conforme o teste e as imagens da aplicação funcionando, pode-se concluir que as condições para *if/then/else* com alteração de visibilidade, troca de texto, troca de valor da variável global estão funcionando como o esperado.

5.3.3 Teste com a calculadora

Por fim, foi realizado um teste final com a calculadora desenvolvida pelo autor Brandt no trabalho anterior a este (BRANDT, 2022), no qual foi verificado que apesar de todas as alterações realizadas no código, na lógica de interpretação do YAIL e também na

estrutura inicial de *Context API*. A aplicação da calculadora continuou a funcionar corretamente.

5.4 Validação dos Requisitos

Os testes funcionais do sistema foram conduzidos pelos próprios autores, utilizando três tipos distintos de teste: testes unitários manuais, aplicação da calculadora desenvolvida pelo autor Brandt e um teste de integração com uma aplicação simulando os primeiros passos de um potencial curso introdutório de lógica e pensamento computacional para alunos do ensino fundamental.

O objetivo desses testes foi validar os requisitos funcionais previamente listados no capítulo de análise. Cada tipo de teste proporcionou uma abordagem diferente para verificar se o sistema atendia aos requisitos estabelecidos e se estava funcionando de acordo com o esperado.

Os testes com componentes isolados permitiram uma análise minuciosa do desempenho e funcionalidade de cada componente individualmente, garantindo que eles operassem corretamente e cumprissem suas respectivas funções.

A aplicação da calculadora desenvolvida pelo autor Brandt proporcionou uma oportunidade de teste mais abrangente, permitindo avaliar a interação entre diferentes componentes em um contexto mais realista. Além disso, foi possível verificar se as implementações de novos componentes e as alterações realizadas nas partes do código compartilhadas por esses componentes não resultaram em comportamentos inesperados. Isso possibilitou verificar se o sistema era capaz de realizar cálculos corretamente e lidar adequadamente com as operações matemáticas.

Esses testes foram essenciais para garantir a qualidade e a funcionalidade do sistema, assegurando que o mesmo estivesse preparado para ser utilizado em um ambiente educacional. Eles forneceram informações valiosas para identificar possíveis problemas, ajustar o funcionamento dos componentes e promover melhorias contínuas no sistema como um todo.

Tabela 23: Resumo dos resultados dos testes realizados

Requisito	Teste realizado		
	Unitário	Integração	App Calculadora
RF1	OK	OK	N/A

RF2	OK	Acionamento sempre depende de ser executado por um Botão	N/A
RF3	OK	N/A	N/A
RF4	OK	Comportou de maneira inesperada quando deveria aparecer ou desaparecer conforme interação do usuário	N/A
RF5	OK	Comportamento inesperado ao aparecer ou desaparecer com base na interação do usuário	N/A
RF6	OK	OK	N/A
RF7	OK	OK	N/A
RF8	OK	OK	N/A
RF9	OK	OK	N/A
RF10	OK	OK	N/A
RF11	OK	A recuperação do arquivo salvo falhou durante os testes	N/A
RF12	OK	N/A	OK
RF13	OK	OK	OK
RF14	OK	OK	N/A

6 Conclusão

O presente trabalho tem como objetivo principal o desenvolvimento evolutivo da ferramenta web LIDEA-W, utilizada para a execução de aplicativos desenvolvidos com o *App Inventor*. Esse objetivo foi alcançado por meio da implementação de componentes atualmente não suportados pela ferramenta.

Para atingir esse objetivo, foi inicialmente realizado um levantamento do estado da arte em relação aos ambientes web para a execução de aplicações desenvolvidas com o *App Inventor*. Assim, uma revisão sistemática da literatura foi conduzida, visando identificar as melhorias necessárias na implementação atual da ferramenta LIDEA-W. Com base nessa análise, foram identificadas as melhorias de padrão de projetos necessárias e as novas funcionalidades a serem suportadas. Também foi realizada uma análise do protocolo de comunicação entre o aplicativo *App Inventor* e o aplicativo *AI2 Companion*, a fim de viabilizar a implementação das novas funcionalidades.

Após essa etapa, foram mapeados os requisitos funcionais dos componentes visuais e não visuais que ainda não haviam sido implementados. Com base nesse mapeamento, foram planejadas iterações para o desenvolvimento dos componentes. Durante a implementação, foi realizada a interpretação do YAIL recebido, atualizando as funções de interpretação já existentes no projeto e ajustando os componentes existentes.

Essas etapas permitiram a implementação bem-sucedida dos componentes necessários, expandindo as funcionalidades da ferramenta LIDEA-W e possibilitando a execução de aplicativos com os componentes previamente não suportados. Nesse sentido, este trabalho contribui para o aprimoramento da experiência de desenvolvimento de aplicativos no *App Inventor*, oferecendo aos desenvolvedores um conjunto mais completo e abrangente de componentes disponíveis na plataforma LIDEA-W.

No entanto, durante a implementação dos componentes, foram identificados problemas arquiteturais que afetam o desempenho dos componentes que envolvem estruturas complexas de blocos, com múltiplos componentes interagindo entre si. A abordagem de desenvolvimento individualizado de cada componente isoladamente resultou em complexidades no acesso às funções, as quais, no *App Inventor*, são acessíveis globalmente.

Essas limitações arquiteturais impactaram a usabilidade e a integração harmoniosa dos componentes desenvolvidos. A falta de um sistema de comunicação e interação eficiente entre os diferentes componentes dificultaram a criação de fluxos de trabalho fluidos e a troca de informações entre eles.

6.1 Trabalhos Futuros

Como sugestão para trabalhos futuros, é recomendado abordar as questões identificadas e buscar soluções para aprimorar a arquitetura dos componentes e melhorar a comunicação entre eles. A revisão da arquitetura permitirá estabelecer um mecanismo eficiente de troca de informações e colaboração entre os componentes, favorecendo o compartilhamento de funções e recursos comuns.

Uma possível melhoria a ser explorada é a utilização de expressões regulares (REGEX) para a conversão mais precisa do código YAIL. Isso garantirá uma tradução mais acurada e confiável, contribuindo para a correta interpretação das funções e a ordenação adequada da execução do código. Além disso, uma alternativa interessante a ser considerada é a implementação de um interpretador léxico, que permitirá uma análise mais detalhada e precisa do código, facilitando a conversão das funções e otimizando a sequência de execução.

Além disso, como continuação das sugestões para trabalhos futuros, seria válido aperfeiçoar os componentes que foram testados isoladamente, garantindo que funcionem conforme o esperado em conjunto com outros elementos. Isso envolveria realizar uma análise aprofundada das interações entre esses componentes e identificar possíveis conflitos ou limitações na integração com outros recursos.

Outro ponto pendente é a lógica de troca de telas dentro do aplicativo, esse requisito funcional foi encontrado durante a análise da versão inicial desenvolvida por Brandt (2022), porém devido aos desafios encontrados na implementação dos componentes levantados e por conta das limitações de entendimento do código YAIL e sua complexidade e repetição de códigos.

Essas melhorias propostas contribuirão para o aprimoramento geral da ferramenta LIDEA-W, proporcionando uma arquitetura mais robusta, uma conversão mais precisa do código YAIL e uma execução mais eficiente dos aplicativos desenvolvidos com o *App Inventor*. Isso resultará em uma experiência de desenvolvimento mais fluida para os usuários, possibilitando a criação de aplicativos complexos e de alta qualidade.

Referências

BRANDT, Darlan. **Plataforma Web para execução de aplicativos Android desenvolvidos com App Inventor**. 2021. 98 f. TCC (Graduação) - Curso de Sistema da Informação, Universidade Federal de Santa Catarina, Florianópolis, 2021. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/233130/TCC-DarlanBrandt-Final.pdf?sequence=1&isAllowed=y>. Acesso em: 10 maio 2022.

FRANÇA, Gustavo. **Ambiente Web integrado com App Inventor para execução de aplicações Android**. 2019. TCC (Graduação) - Curso de Sistema da Informação, Universidade Federal de Santa Catarina, Florianópolis, 2019. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/202667/TCC.pdf?sequence=1&isAllowed=y>. Acesso em: 03 nov. 2022.

MIT APP INVENTOR (org.). Disponível em: <https://appinventor.mit.edu/>. Acesso em: 25 maio 2022.

MIT APP INVENTOR (org.). **How Does my Android or iOS Device Connect Over Wi-Fi?**. Disponível em: <https://appinventor.mit.edu/explore/support/explain-wifi-connection>. Acesso em: 22 out. 2022.

INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). **Censo Escolar, 2021**. Brasília: MEC, 2022. Disponível em: https://download.inep.gov.br/censo_escolar/resultados/2021/apresentacao_coletiva.pdf. Acesso em: 25 maio de 2022.

BANKS, Alex *et al.* **Learning React**. 2. ed. United States Of America: O'reilly Media, 2020

FACEBOOK INC. **React – Uma biblioteca JavaScript para criar interfaces de usuário**. Reactjs.org, 2021. Disponível em: <https://pt-br.reactjs.org/>. Acesso em: 16 out. 2022.

CODECADEMY. **React – The virtual dom**. Codecademy Team, 2022. Disponível em: <https://www.codecademy.com/article/react-virtual-dom>. Acesso em: 16 out. 2022.

BRASIL. GOV. . **Aprovado parecer que define normas sobre o ensino de computação na educação básica.** 2022. Disponível em: <https://www.gov.br/pt-br/noticias/noticias/educacao-e-pesquisa/10/aprovado-parecer-que-define-normas-sobre-o-ensino-de-computacao-na-educacao-basica>. Acesso em: 16 out. 2022.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC). **Diretrizes para ensino de Computação na Educação Básica.** 2019. Disponível em: <https://www.sbc.org.br/educacao/diretrizes-para-ensino-de-computacao-na-educacao-basica>. Acesso em: 16 out. 2022.

BRACKMANN, C. P. **Computacional / Computação na Educação Básica.** 2022. Disponível em: <https://www.computacional.com.br>. Acesso em: 16 out. 2022.

SOCIEDADE BRASILEIRA DE COMPUTAÇÃO (SBC). **Referenciais de Formação em Computação: Educação Básica.** 2017. Disponível em: <https://www.sbc.org.br/images/ComputacaoEducacaoBasica-versaofinal-julho2017.pdf>. Acesso em: 16 out. 2022.

SCHILLER, J. et al. **Live Programming of Mobile Apps in App Inventor.** Proceedings of the 2nd Workshop on Programming for Mobile & Touch - PROMOTO '14, 2014. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/2688471.2688482>. Acesso em: 17 out. 2022

GNU. **Kawa: The Kawa Scheme language.** Disponível em: <https://www.gnu.org/savannah-checkouts/gnu/kawa/index.html>. Acesso em: 17 out. 2022.

SAYYAPARAJU, Vedha. **User-Designed Background Tasks in App Inventor.** 2014. 70 f. Tese (Doutorado) - Curso de Engineering In Electrical Engineering And Computer Science, Massachusetts Institute Of Technology, Cambridge, 2015. Acesso em: 17 out. 2022.

ANTONIO e LEMOS. **As competências do professor do século XXI: a tecnologia como instrumento de apoio.** 2015. Disponível em: <https://periodicos.unisanta.br/index.php/hum/article/view/333/365>. Acesso em: 21 out. 2022.

GARLET, et al. **Uma proposta para o Ensino de Programação de Computadores na Educação Básica.** 2016. Disponível em: https://repositorio.ufsm.br/bitstream/handle/1/12961/TCCG_SIFW_2016_GARLET_DANIELA.pdf?sequence=1&isAllowed=y. Acesso em: 21 out. 2022.

COMPUTAÇÃO NA ESCOLA. **Computação na Escola**, 2022. Disponível em: <https://computacaonaescola.ufsc.br/quem-somos/>. Acesso em: 21 out. 2022.

CLARKE, N; ABELSON, H. **Democratizing Cross-Platform Mobile Development with MIT App Inventor**, 2018. Disponível em: https://drive.google.com/file/d/1xCwT3fpXdAupVI5oE5_-dcmOaw0z149H/view>. Acesso em: 1 nov. 2022.

WANG, Katherine. **Creating a Database Log History Page for Cloud Components in App Inventor to Support Software Development Education**, 2018. Disponível em: https://drive.google.com/file/d/0BzGL9U0OLqkQUdxQ281Z3B6UIA4VE42UzVNZ1hGb0pKUkx3/view?resourcekey=0-175_LmOUo2CCm6BCGQIJ3w>. Acesso em: 1 nov. 2022.

GONÇALVES, Pablo. **O que é programação em blocos?**, 2022. Disponível em: <http://professorpablo.com/o-que-e-programacao-em-blocos/>>. Acesso em: 3 nov. 2022.

MEDEIROS, Giselle A.; BERGMANN, Juliana C; WANGENHEIM, Christiane G. **Práticas pedagógicas com o desenvolvimento de aplicativos para dispositivos móveis por estudantes da educação básica**. 2020. Disponível em: <http://www.periodicos.ulbra.br/index.php/txra/article/view/5052/3670>. Acesso em: 21 out. 2022.

DOMINGUEZ, J.; SPERTUS, E. App Inventor Developer Overview. **MIT App Inventor Sources**, 2021. Disponível em: <https://docs.google.com/document/u/0/d/1hIvAtbNx-eiIJcTA2LLPQOawctiGIpnnt0AvfgnKBok/pub>>. Acesso em: 21, out. 2022.

GWT PROJECT. **GWT Overview**, 2022. Disponível em: <https://www.gwtproject.org/overview.html>>. Acesso em: 21 out. 2022

LYE, S. Y., KOH, J. H. L. **Review on teaching and learning of computational thinking through programming: What is next for K-12?**. Computers in Human Behavior, v. 41(C), p. 51-61. 2014. Disponível em: <https://www.semanticscholar.org/paper/Review-on-teaching-and-learning-of-computational-is-Lye-Koh/64b5f719a6f7bff3c58e620d859d7dd5a3d3fdc1>>. Acesso em: 03 nov. 2022

GARCIA, L. F. F. **RPC - Remote Procedure Call**. Repositório de Documentos Técnicos do Grupo de Redes - UFRGS, 2002. Disponível em: <<http://penta.ufrgs.br/rc952/trab1/rpc.html#rpc>> . Acesso em: 03 nov. 2022.

NODE.JS. **Sobre | Node.js**, 2022. Disponível em: <<https://nodejs.dev/en/about/>>. Acesso em: 01 nov. 2022.

BUCKLER, Craig. **Node.js: Novice to Ninja**. [S. L.]: Sitepoint Pty. Ltd, 2022.

SILVA, E. D.; MENEZES, E. **Metodologia da Pesquisa e Elaboração de Dissertação**. [S.l.: s.n.], 2005

MICROSOFT. **Estilo de arquitetura de microsserviço - Azure Architecture Center**. Disponível em: <<https://learn.microsoft.com/pt-br/azure/architecture/guide/architecture-styles/microservices>> . Acesso em: 8 nov. 2022.

CSTA. **Computer Science Teachers Association**, 2019. Disponível em: <<https://www.csteachers.org/page/about-csta-s-k-12-nbsp-standards>>. Accessed 8 Nov. 2022.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. **Systematic Mapping Studies in Software Engineering**. Proc. of the 12th Int. Conference on Evaluation and Assessment in Software Engineering, Bari, Italy, pp. 68-77, 2008

MENDONÇA, R. A. R. de. **Levantamento de requisitos no desenvolvimento ágil de software**. Semana da Ciência e Tecnologia da PUC Goiás, p. 12, 2014.

SANJUGTHA, Shoba. **The Art of Exploratory Testing**. StickyMinds, 2020. Disponível em: <<https://www.stickyminds.com/article/art-exploratory-testing.>>. Accessed 8 Nov. 2022.

SCHWABER, Ken. **Scrum Guide**, 2020. Disponível em: <<https://scrumguides.org/scrum-guide.html>>. Acessado 29 de novembro de 2022.

DAVIDSON, D. **Why do we use Story Points for Estimating?**, 2014. Disponível em: <<https://www.scrum.org/resources/blog/why-do-we-use-story-points-estimating>>.

SCRUM.ORG. **What is a Product Backlog?**, 2019. Disponível em: <<https://www.scrum.org/resources/what-is-a-product-backlog>>. Acesso em: 12 dez. 2022.

Apêndice I - Código-Fonte

O código-fonte da aplicação desenvolvida neste trabalho se encontra disponível no repositório: <https://codigos.ufsc.br/gqs/lidea-w>

Apêndice II - Artigo

Aprimoramento de uma ferramenta de Live Debugging Web para aplicativos desenvolvidos com o App Inventor

Guilherme Alexandrino Proença e Rafaela Bernardo Rodrigues

Departamento de Informática e Estatística (INE) Universidade Federal de Santa Catarina (UFSC) - Florianópolis, SC - Brazil

guiiaalexandrino@gmail.com / rafabernardo.r@gmail.com

Abstract. *Developing a mobile application is not a simple task, as it requires technical programming knowledge and specific software. In order to overcome this technical barrier, applications have started to emerge to facilitate app development and introduce new users to programming. An example of this is App Inventor, a free tool aimed at promoting computational thinking through app development using visual programming blocks. However, using App Inventor requires the user to have a smartphone or use an emulator, which in turn requires a certain level of computer capability to visualize the app during development. This requirement can become a restriction in public schools where many students do not have smartphones. As a proof of concept, the development of a web application was initiated to meet the need for using a mobile phone to visualize the progress of the project developed in App Inventor. Therefore, this work has the objective to develop new features for this tool.*

Resumo. *Desenvolver um aplicativo para celular não é uma tarefa simples, pois é necessário ter conhecimento técnico em programação e possuir softwares específicos. Dessa forma, buscando romper com essa barreira técnica, começaram a surgir aplicações para facilitar o desenvolvimento de aplicativos e introduzir novos usuários à programação. Um exemplo disso é o App Inventor, uma ferramenta gratuita que tem como propósito promover o pensamento computacional por meio do desenvolvimento de aplicativos, utilizando programação com blocos visuais. No entanto, para se visualizar o aplicativo em tempo de desenvolvimento, o do App Inventor requer que o usuário possua um smartphone ou utilize um emulador, que, por sua vez, demanda um certo nível de capacidade do computador. Essa exigência pode se tornar uma restrição em escolas públicas onde muitos alunos não possuem celulares e os computadores são precários. Com isso, foi iniciado, como prova de conceito, o desenvolvimento de uma aplicação web para suprir a necessidade do uso do celular para conseguir visualização do andamento do projeto desenvolvido no App Inventor. Assim, este trabalho busca desenvolver novas funcionalidades para essa ferramenta.*

1. Introdução

O *App Inventor* é uma aplicação web de código aberto desenvolvida pelo *Massachusetts Institute of Technology (MIT)* (MIT APP INVENTOR, 2022). Seu propósito é promover o pensamento computacional por meio do desenvolvimento de aplicativos móveis

amadores, sem a necessidade de se aprender uma linguagem textual, pois utiliza programação com blocos visuais. Dessa forma, a ferramenta *App Inventor* democratiza o aprendizado do pensamento computacional e estimula o interesse de crianças e adolescentes na área de tecnologia. Com mais de um milhão de visitantes mensais de 195 países diferentes (MIT APP INVENTOR, 2022), a aplicação passou a ser usada em inúmeros projetos e iniciativas que buscam inserir o ensino da computação na educação básica em diversos países e no contexto da educação brasileira não foi diferente.

Atualmente, a depuração dos aplicativos desenvolvidos com o *App Inventor* necessita do suporte de um aplicativo chamado *MIT AI2 Companion*. Através desse app é possível realizar a conexão a um dispositivo (smartphone ou tablet) que executará o aplicativo em desenvolvimento, em tempo real, ou seja Live Debugging. A conexão entre o aplicativo *MIT AI2 Companion* e o *App Inventor* pode ser feita via Wi-Fi, no entanto ambos devem estar conectados à mesma rede local. No entanto, o Censo Escolar do ano de 2021, realizado pelo INEP, indicou que menos de 40% das escolas do ensino fundamental da rede pública possuem acesso a um computador e acesso à internet para ensino e aprendizagem (INEP, 2021).

Considerando então o processo de conexão para a etapa de depuração e o cenário atual do resultado da pesquisa feita pelo INEP, tem-se uma limitação de uma das etapas fundamentais para quem está iniciando a prática do desenvolvimento que é visualizar, testar e entender o que foi criado. Além da falta de computadores com o acesso a internet, a necessidade de um smartphone é um agravante pois não faz parte da realidade de muitas das crianças e adolescentes de escola pública como tem sido percebido nas aplicações voltadas ao ensino do pensamento computacional realizadas pela iniciativa Computação na Escola³ no Brasil.

Para tentar diminuir a complexidade e as diversas limitações para a utilização do *App Inventor*, foi iniciado o desenvolvimento de uma ferramenta *web*, chamada LIDEA-W, para a visualização do aplicativo em desenvolvimento e sua depuração sem a necessidade de criação de uma máquina virtual, utilização de emuladores ou utilização de um *smartphone* (BRANDT, 2021).

No entanto, o trabalho realizado por Brandt (2021) sugere várias modificações que não foram viáveis de serem realizadas devido à sua complexidade e limitações de tempo. Essas modificações incluem a conversão de certos elementos não visuais do *App Inventor* em novos componentes que podem aproveitar o *hardware* disponível em computadores, como o microfone, as caixas de som e a câmera. A inclusão de tais componentes acrescenta funcionalidades que tornariam o ensino do uso da ferramenta mais divertido para os estudantes (BRANDT, 2021).

Com isso, este trabalho propõe a evolução desta aplicação para que seja possível a implantação do sistema *web* na nuvem de forma a atenuar as dificuldades da utilização do *App Inventor*, sobretudo em escolas públicas com alunos de baixa renda que podem não possuir acesso a *smartphones*.

2. Conceitos importantes

Neste capítulo, serão discutidos os princípios fundamentais relacionados à elaboração deste trabalho. Serão abordados temas como *App Inventor* e suas peculiaridades, *AI2*

³ <https://computacaonaescola.ufsc.br/>

Companion e a computação na escola.

2.1 Computação na escola

As mudanças políticas, econômicas e sociais que vêm ocorrendo nos últimos anos são responsáveis por alterar de forma significativa a sociedade. Fenômenos como a Globalização e a chamada Era da Informática introduziram novos desafios à sociedade atual (ANTONIO e LEMOS, 2015).

Nesse cenário de evolução tecnológica, é normal que a demanda por profissionais capacitados também aumente e que estes devam possuir habilidades relacionadas ao pensamento computacional, mesmo que não trabalhem em áreas ligadas diretamente à computação. Entretanto, é importante que esse conhecimento seja desenvolvido o quanto antes, para facilitar a familiarização ao conhecimento computacional e o aprendizado de novas formas de resolver problemas.

De acordo com GARLET et al. (2016), no Brasil o aprendizado da Informática ou mais especificamente da programação, está quase restrito apenas às pessoas que buscam por essas áreas de conhecimento. Entretanto, surgem iniciativas como o Computação na Escola, com o intuito de alterar esse cenário.

A Iniciativa Computação na Escola, do Departamento de Informática e Estatística da Universidade Federal de Santa Catarina, tem como objetivos: motivar e facilitar o ensino da computação para todos os alunos do Ensino Fundamental e Médio no Brasil, capacitar professores para esse ensino, compartilhar relatos de experiências e recursos a fim de criar uma comunidade de prática e aumentar o número de formados na área de computação - buscando elevar a representatividade de mulheres e minorias (COMPUTAÇÃO NA ESCOLA, 2022).

2.2 App Inventor

O *App Inventor* é uma ferramenta de programação visual baseada em blocos, que possibilita a criação de aplicativos para sistemas Android por meio de uma interface acessada via navegador *web*. Lançado pela Google em 2010, atualmente é mantido pelo MIT (*Massachusetts Institute of Technology*). Seu principal objetivo é democratizar o desenvolvimento de *software*, capacitando todas as pessoas, especialmente os jovens, a transitar do consumo para a criação de tecnologia.

De acordo com Lye E Koh (2014) a democratização do *App Inventor* se deve porque ambientes de programação baseados em blocos simplificam a programação, já que se fundamentam em um processo de arrastar e encaixar blocos de comando. Isso reduz as ações de programar em si, fazendo com que os estudantes foquem na lógica, sem se preocupar com a sintaxe da programação, assim facilitando o aprendizado.

A programação em blocos tem ganhado destaque nos últimos anos, de acordo com Gonçalves (2022). Isso se deve à sua metodologia, que dispensa o uso de uma linguagem de programação convencional, eliminando a necessidade de conhecimento de sintaxe e regras. Em vez de linhas de código, são utilizados blocos com formas e cores específicas, que auxiliam na diferenciação de suas funções. A organização sequencial desses blocos possibilita a realização de tarefas e a resolução de problemas.

2.3 AI2 Companion

O *MIT AI2 Companion* é um aplicativo para Android e IOS que permite realizar a conexão entre um *smartphone* ou *tablet* via USB ou via Wi-Fi com a aplicação desenvolvida no *App Inventor*, com o objetivo do usuário conseguir acompanhar a evolução do seu aplicativo desenvolvido no *App Inventor*. Ou seja, o usuário irá conseguir executar o código desenvolvido e com isso testar sua aplicação, visualizando o resultado em tempo real, em seu *smartphone* ou *tablet*.

Segundo SCHILLER, et al. (2014) a principal função de visualizar o aplicativo desenvolvido em tempo real, seria o *live debugging*, que permite obter uma forma de visualizar imediatamente as alterações realizadas no programa desenvolvido, assim, o programador consegue entender e experimentar o aplicativo criado. Conforme FRANÇA (2019), vale lembrar que o usuário deve ter um aparelho Android ou IOS com o *Companion* instalado e deve estar conectado na mesma rede wi-fi que o computador onde a aplicação está sendo desenvolvida.

A conexão entre o *smartphone* e um computador só é possível através da conexão com o *MIT RendezVous Server*, e tanto o *smartphone* quanto o computador necessitam estar conectados na mesma rede. A conexão é realizada em seis passos (BRANDT, 2021):

1. Computador faz *check-in* com um código de seis caracteres gerado pelo *MIT AI2 Companion*;
2. *MIT RendezVous Server* registra o endereço IP do computador;
3. Dispositivo móvel faz *check-in* com um código de 6 caracteres;
4. *MIT RendezVous Server* registra o endereço IP do dispositivo móvel;
5. *MIT RendezVous Server* informa os endereços IP dispositivo móvel e o computador, que possuem a mesma chave de seis caracteres, para o outro;
6. Conexão estabelecida e ambos passam a se comunicar via IP.

Por fim, o *Companion* também é responsável por checar se houve modificação no código ou não. Caso tenham ocorrido mudanças, ele exibe as alterações em tempo real na tela. Para conseguir realizar tal tarefa ele verifica se o código recebido é igual ao que está em cache, se for diferente ele atualiza.

3. Análise e projeto

Neste capítulo é apresentado o levantamento e análise dos requisitos, seguido da estimativa, planejamento e desenvolvimento da solução técnica.

3.1 Levantamento de análise de requisitos

O levantamento de requisitos é considerado o início para toda a atividade de desenvolvimento de *software*, sendo considerado um papel importante na construção de um sistema de informação (MENDONÇA, 2014). Para realizar a análise e levantamento dos requisitos desta plataforma foram realizados análise do código já desenvolvido por Brandt (2021) e testes exploratórios nos componentes existentes na aplicação do *App Inventor*, já que

segundo Sanjugtha (2020) o teste exploratório consiste em examinar todas as camadas do aplicativo enquanto pensa em todas as possibilidades de casos de uso. Cada um dos componentes visuais e não-visuais do ambiente de desenvolvimento do *App Inventor* foi explorado. A partir desses testes exploratórios, foram identificados os requisitos funcionais, para que possam ser desenvolvidos na evolução da ferramenta LIDEA-w proposta neste trabalho.

3.2 Requisitos Funcionais

3.2.1 Componentes de interface do usuário

Na tabela abaixo é apresentado os requisitos funcionais referentes aos componentes de *user interface* do *App Inventor* a serem implementados na plataforma LIDEA-W

RF1	Desenvolver o componente <i>ListView</i>
Permite adicionar uma lista de conteúdos, texto ou imagens	
RF2	Desenvolver o componente <i>Notifier</i>
Permite mensagens de alerta	
RF3	Desenvolver o componente <i>WebView</i>
Permite visualizar um site externo a aplicação.	

Tabela 1: Componentes de interface do usuário

3.2.2 Componentes de mídia

Na tabela abaixo é apresentado os requisitos funcionais referentes aos componentes de mídia do *App Inventor* a serem implementados na plataforma LIDEA-W.

RF4	Desenvolver o componente <i>Camcorder</i>
Permite usar a câmera para gravar vídeos	
RF5	Desenvolver o componente <i>Camera</i>
Permite tirar uma foto utilizando a câmera do dispositivo.	
RF6	Desenvolver o componente <i>ImagePicker</i>
Permite selecionar uma foto da minha galeria de imagens	
RF7	Desenvolver o componente <i>Image</i>
Permite visualizar arquivos de imagem	

RF8	Desenvolver o componente <i>VideoPlayer</i>
Permite visualizar vídeos existentes no dispositivo.	
RF9	Desenvolver o componente <i>Player</i>
Permite executar um arquivo de áudio do dispositivo.	
RF10	Desenvolver o componente <i>Sound</i>
Permite executar um arquivo de áudio com poucos segundos.	

Tabela 2: Componentes de mídia

3.2.3 Componentes de armazenamento

Na tabela abaixo é apresentado o requisito funcional referente ao componente de armazenamento do *App Inventor* a ser implementado na plataforma LIDEA-W.

RF11	Desenvolver o componente <i>TinyDB</i>
Permite armazenar dados usando chave e valor	

Tabela 3: Componentes de armazenamento

3.2.4 Implementações na lógica de interpretação do código Yail

Na tabela abaixo é apresentado os requisitos funcionais referentes às alterações necessárias na lógica da aplicação responsável por interpretar o código YAIL.

RF12	Refatoração das condicionais
Permite os comandos de condição dos blocos do <i>App Inventor</i> sendo interpretados dentro da aplicação	
RF13	Refatoração variáveis globais
Permite utilizar variáveis globais do <i>App Inventor</i> sendo interpretadas e alteradas dentro da aplicação	
RF14	Alterar visibilidade dos componentes
Permite alterar a visibilidade dos componentes a partir de ações de blocos.	

Tabela 4: Implementações na lógica de interpretação

4. Implementação

Este capítulo descreve a implementação do sistema web, baseado na análise e levantamento dos requisitos realizados. São abordados aspectos essenciais da implementação como o desenvolvimento, testes funcionais e validação de requisitos.

4.1 Desenvolvimento

Nesta seção, a implementação do projeto é apresentada, bem como as dificuldades encontradas ao longo do desenvolvimento.

4.1.1 ListView

O componente *ListView* exibe uma lista de itens na interface do *App Inventor*, com opção de busca pelo nome do item. Ele utiliza os componentes *List*, *ListItem*, *ListAvatar*, *ListItemText*, *InputLabel* e *Input* da biblioteca *MaterialUI* para o React. O desafio foi transmitir o valor do item selecionado para uma variável global, o que foi resolvido utilizando o recurso *Context* API do React. A busca reativa foi implementada utilizando o método *filter* de arrays do JavaScript. A implementação visual do componente foi simples, com base na documentação do *Material UI*.

4.1.2 Notifier

O componente *Notifier* apresentou um desafio de acionamento por uma função chamada por outro componente, *Button*. A arquitetura do projeto inicialmente não permitia o acesso a funções associadas a componentes isolados. Para contornar isso, utilizou-se o componente *Dialogs* do *Material UI*, que utiliza o conceito de *Portal* para renderizar elementos em um nó *DOM* externo. No entanto, isso tornou o componente inacessível de acordo com a arquitetura planejada. Para resolver o problema, foi utilizado o *Context* API do React, permitindo o compartilhamento de dados entre componentes dentro do escopo do contexto.

4.1.3 WebViewer

A implementação do *WebViewer* utilizou o elemento *HTML* chamado *iframe*, que possibilita o carregamento de outra página *HTML* dentro do *DOM*. De forma essencial, o elemento *iframe* permite a inclusão de uma página da web dentro da página principal. A implementação do *WebViewer* foi realizada de maneira simples, sem apresentar nenhum problema significativo.

4.1.4 Camcorder

O componente *Camcorder* permite a gravação de vídeos utilizando a câmera do dispositivo no *App Inventor*. Possui um método chamado *RecordVideo*, que inicia a gravação e emite um evento *AfterRecording* após a conclusão, fornecendo o caminho do vídeo gravado. Para sua implementação, foi utilizada a biblioteca externa *React-WebCam*, que simplifica a manipulação de imagens e vídeos da webcam. A dificuldade encontrada envolveu a transformação dos dados da mídia gravada em formato binário para transmiti-los ao componente *VideoPlayer*. A documentação do *React-WebCam* foi consultada para entender a

sequência correta de métodos e a transformação dos dados. Foi necessário criar um Context API global para transmitir os dados e alterações para o componente *VideoPlayer*.

4.1.5 Camera

O componente Camera do *App Inventor* permite tirar fotos usando a câmera do dispositivo. Ao acionar o método *TakePicture*, uma foto é capturada e o evento *AfterPicture* é emitido, fornecendo o caminho da imagem para exibição na interface. Para seu funcionamento adequado, o componente trabalha em conjunto com o *TinyDb* e o componente Image. O *TinyDb* armazena os dados binários da imagem, enquanto o componente Image exibe a foto com base no caminho fornecido. A comunicação entre esses componentes é estabelecida por meio do *Context* API do React, garantindo o compartilhamento dos dados necessários para a exibição da imagem capturada.

4.1.6 VideoPlayer

O componente *VideoPlayer* reproduz vídeos no App Inventor e possui métodos para iniciar, pausar, pular e recomeçar a reprodução. Ele utiliza o elemento `<video>` nativo do HTML e não requer bibliotecas externas. O componente pode funcionar de duas maneiras: recebendo a URL do vídeo diretamente ou trabalhando em conjunto com o Camcorder, que grava o vídeo. Quando recebe o vídeo do Camcorder, utiliza os dados compartilhados pelo Context API. O desafio principal foi transmitir as ações de manipulação do vídeo, como iniciar, pausar e recomeçar. Para isso, foi atribuído um ID ao elemento de vídeo no HTML e as ações foram manipuladas através de alterações do DOM utilizando JavaScript.

4.1.7 Image

Para implementar o componente de imagem no projeto, foi necessário realizar um estudo aprofundado sobre o funcionamento do *App Inventor* e do *AI2 Companion*, especialmente em relação ao envio de arquivos entre as aplicações. Foi realizada uma engenharia reversa para entender o processo de envio de imagens e a implementação específica da requisição PUT, que desempenha um papel fundamental nesse contexto.

Utilizando a guia de rede do navegador, foi identificado o uso do método HTTP PUT para enviar arquivos binários. No *App Inventor*, essa requisição PUT é utilizada para enviar as imagens inseridas pelos usuários para o servidor. Essas imagens são tratadas como "assets" e são essenciais para o funcionamento do aplicativo.

Por meio da análise minuciosa do código-fonte do *App Inventor*, foi possível compreender a implementação específica dessa requisição PUT e como as imagens eram manipuladas nesse contexto. Esse conhecimento detalhado dos mecanismos e protocolos envolvidos foi fundamental para garantir a correta implementação do componente de imagem no projeto.

4.1.8 Player

O App Inventor possui um componente chamado Player, que é responsável por reproduzir áudios, especialmente músicas. Este componente não é visível na interface do

usuário e não possui uma interface gráfica própria. Os botões de pausar, iniciar e recomeçar devem ser adicionados pelo usuário, juntamente com as chamadas para os métodos correspondentes de manipulação do áudio.

A implementação do componente Player utilizou o elemento `<audio>` nativo do HTML, sem a necessidade de bibliotecas externas. Para ocultar o elemento na interface, foi necessário definir a visibilidade padrão como "hidden", garantindo que apenas o áudio seja reproduzido sem exibir a interface.

O principal desafio no desenvolvimento do componente foi transmitir os comandos do usuário para o elemento HTML. Para isso, foi utilizada uma abordagem semelhante à do componente VideoPlayer, onde os comandos são executados por meio da manipulação do DOM, invocando as funções nativas do elemento `<audio>`.

4.1.9 Sound

O outro componente responsável por reproduzir áudio dentro do App Inventor é o Sound. Ao contrário do Player o Sound é indicado para áudios curtos. Ele também é um componente não visível, por isso não possui nenhum tipo de interface gráfica na aplicação. Seu funcionamento a estrutura do componente é exatamente a mesma do Player.

A diferença entre os dois componentes é que enquanto o Player possui os métodos: Pause, Start, Stop o Sound possui Pause, Resume e Start. Na prática usando esse componente não podemos notar diferença entre eles, os desafios para o desenvolvimento desse componente foram os mesmos do anterior.

4.1.10 Image Picker

O componente *ImagePicker* é um botão especializado que permite ao usuário selecionar uma imagem da galeria do dispositivo. Ao ser acionado, o usuário pode escolher a imagem desejada, e o componente armazena o caminho do arquivo da imagem selecionada na propriedade "Selection". A implementação do componente utiliza o botão da biblioteca Material UI combinado com o elemento HTML `<input>` do tipo "file", proporcionando uma funcionalidade simples e eficiente.

No entanto, um desafio encontrado foi atualizar o estado das mídias e tornar a imagem selecionada acessível para o componente de imagem. Para resolver isso, foi criada uma função chamada "updateMedias" no Contexto de Conteúdo (*ContentContext*) que adiciona a nova imagem ao array de mídias e atualiza as propriedades do componente correspondente. Essa abordagem permitiu a atualização eficiente do estado das mídias e das propriedades do componente de imagem, superando o desafio de forma adequada.

4.1.11 TinyDB

A implementação do *TinyDB* no LIDEA-W utilizou o *IndexedDB*, uma API nativa dos navegadores, juntamente com a biblioteca IDB. Essa combinação permitiu o armazenamento persistente e confiável dos dados no navegador, proporcionando uma experiência contínua aos usuários. O *TinyDB* utiliza tags e valores para armazenar e recuperar dados, onde as tags servem como identificadores dos itens armazenados.

Além disso, para garantir o acesso adequado às funções do *TinyDB*, foi adotada a abordagem de *Context*, permitindo que essas funções estejam disponíveis para todos os componentes que precisam realizar o armazenamento de informações. Essa estratégia facilita o acesso e uso das funcionalidades do *TinyDB* por diferentes partes do aplicativo.

4.1.12 Refatoração das condicionais

A refatoração das condicionais foi uma etapa muito importante e desafiadora no desenvolvimento deste trabalho, pois foi necessário realizar melhorias na lógica de interpretação do código Yail. A complexidade da sintaxe de interpretação dentro do YAIL, aumenta à medida que adicionamos métodos, ações dentro dos blocos condicionais, resultando em uma difícil tarefa na hora de traduzir o código para o Javascript.

Para conseguir contornar essa dificuldade, foi necessário refatorar os métodos específicos de tradução e adicionar novas técnicas mais precisas de tradução. A principal estratégia adotada para facilitar esse processo de tradução foi recorrer ao uso do Regex, para identificar padrões e assim melhorar a capacidade de tradução do código.

4.1.13 Refatoração variáveis globais

A utilização de variáveis desempenha um papel crucial em uma aplicação, pois permite o armazenamento de valores resultantes de operações. Essa funcionalidade é fundamental para exibir informações na interface do programa e também desempenha um papel vital na construção do algoritmo.

Assim, o primeiro passo consistiu em adicionar um novo valor ao dicionário de interpretação de YAIL, a fim de detectar quando uma string recebida continha uma variável global e, assim, ser capaz de identificar o seu valor inicial. Em seguida, foi necessário desenvolver um método capaz de alterar as propriedades do componente dentro do *Context API* da aplicação, com o objetivo de modificar o valor da variável global de acordo com a ação executada pelo usuário na interface.

4.1.14 Alterar visibilidade dos componentes

É uma prática bastante comum em qualquer aplicação permitir que por meio de uma ação, um componente possa ser exibido ou ocultado. A fim de implementar essa funcionalidade no projeto, foi necessário analisar a forma como as propriedades estavam sendo armazenadas no *Context API* padrão da aplicação.

Foi constatado que, por padrão, os componentes montados dentro da interface do *App Inventor* eram renderizados sem nenhuma indicação de sua visibilidade. Então foi necessário corrigir esse ponto e depois criar um método capaz de alterar essa visibilidade de cada componente dentro do Contexto API.

4.2 Testes Funcionais

Para realizar os testes de integração dos componentes desenvolvidos, foram realizadas duas formas de testes. A primeira consiste em uma aplicação-modelo no App Inventor que utilizou os componentes *ListView*, *Image*, *Sound*, *Player*, *VideoPlayer* além de também testar as alterações executadas na lógica de interpretação das condições.

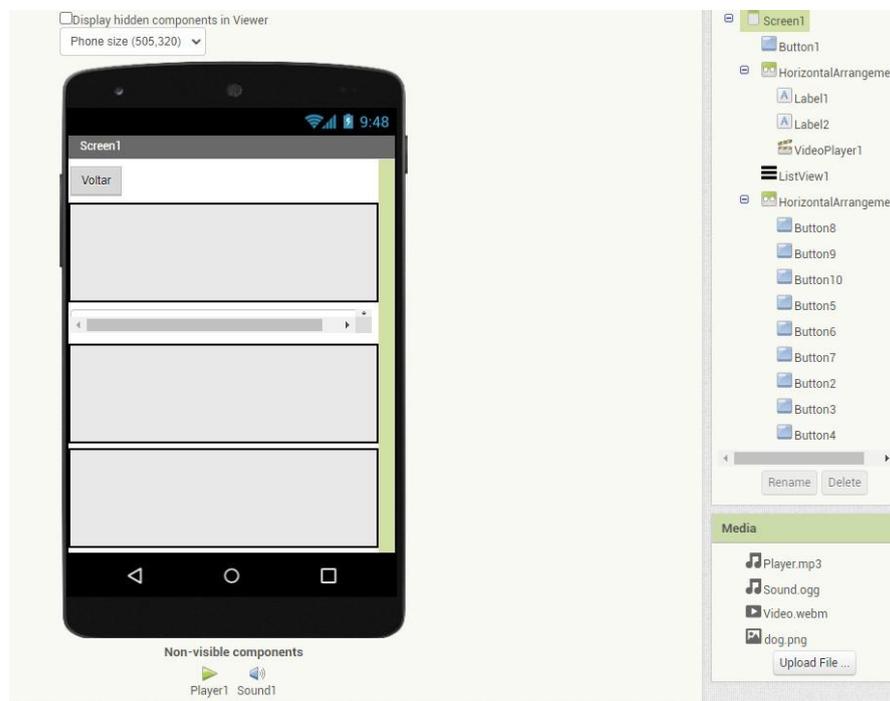


Figura 1: Aplicação modelo gerado no App Inventor

A segunda forma de testar foi realizando testes unitários com os demais componentes desenvolvidos neste trabalho, ou seja, foram testados de maneira isolada, sem estar em um contexto que interagiam com outros componentes totalmente diferentes dos responsáveis pela sua ativação e funcionamento. Os componentes submetidos a esse teste foram: Notifier, WebViewer, Image Picker, TinyDB, Camera, Camcorder, Calculadora desenvolvida por BRANDT (2022).

4.4 Validação dos Requisitos

O objetivo dos testes foi validar os requisitos funcionais. Cada tipo de teste proporcionou uma abordagem diferente para verificar se o sistema atendia aos requisitos estabelecidos e se estava funcionando de acordo com o esperado.

O testes da aplicação modelo permitiu uma análise de como os componentes se comportaram interagindo com outros componentes fora do seu contexto habitual, enquanto os testes isolados permitiram uma análise minuciosa do desempenho e funcionalidade de cada componente individualmente, garantindo que eles funcionassem corretamente e cumprissem suas funções.

Já o teste realizado com a calculadora feita por BRANDT (2022) permitiu verificar se as implementações dos novos componentes e as alterações realizadas nas partes de código compartilhadas por esses componentes não resultaram em comportamentos inesperados.

5. Conclusão

O objetivo principal deste trabalho foi o desenvolvimento evolutivo da ferramenta web LIDEA-W, utilizada para a execução de aplicativos desenvolvidos com o *App Inventor*. Esse objetivo foi alcançado por meio da implementação de componentes atualmente não suportados pela ferramenta.

Para que este objetivo principal fosse atingido, foi necessário identificar melhorias no padrão de projetos e formas de implementar as novas funcionalidades a serem suportadas, sem comprometer o que já tinha sido feito por Brandt (2022). Também foi realizada uma análise do protocolo de comunicação entre o aplicativo *App Inventor* e o aplicativo *AI2 Companion*, a fim de viabilizar a implementação das novas funcionalidades.

Após essa etapa, os requisitos funcionais dos componentes visuais e não visuais que ainda não haviam sido implementados foram identificados e analisados. Com base nessa análise, foram planejadas iterações para o desenvolvimento desses componentes. Durante a implementação, houve uma interpretação do YAIL recebido, o que envolveu atualizar as funções de interpretação já existentes no projeto e ajustar os componentes existentes de acordo.

Essas etapas permitiram a implementação bem-sucedida dos componentes necessários, expandindo as funcionalidades da ferramenta LIDEA-W e possibilitando a execução de aplicativos com os componentes previamente não suportados. Nesse sentido, este trabalho contribui para o aprimoramento da experiência de desenvolvimento de aplicativos no *App Inventor*, oferecendo aos desenvolvedores um conjunto mais completo e abrangente de componentes disponíveis na plataforma LIDEA-W.

Contudo, durante a implementação dos componentes, foram identificados problemas arquiteturais que impactam o desempenho dos componentes com estruturas complexas de blocos e interações entre vários componentes. A abordagem de desenvolver cada componente individualmente trouxe complicações no acesso às funções que, no *App Inventor*, são globalmente acessíveis. LIDEA-W.

As limitações arquiteturais afetaram a usabilidade e integração dos componentes, dificultando a criação de fluxos de trabalho fluidos e a troca de informações entre eles. A ausência de um sistema eficiente de comunicação e interação entre os componentes foi um obstáculo significativo observado neste trabalho.

Referências

BRANDT, Darlan. **Plataforma Web para execução de aplicativos Android desenvolvidos com App Inventor**. 2021. 98 f. TCC (Graduação) - Curso de Sistema da Informação, Universidade Federal de Santa Catarina, Florianópolis, 2021. Disponível em: <https://repositorio.ufsc.br/bitstream/handle/123456789/233130/TCC-DarlanBrandt-Final.pdf?sequence=1&isAllowed=y>. Acesso em: 02 julho 2023.

MIT APP INVENTOR (org.). Disponível em: <https://appinventor.mit.edu/>. Acesso em: 02 julho 2023.

MIT APP INVENTOR (org.). **How Does my Android or iOS Device Connect Over Wi-Fi?**. Disponível em:

<<https://appinventor.mit.edu/explore/support/explain-wifi-connection>>. Acesso em: 03 julho 2023.

INSTITUTO NACIONAL DE ESTUDOS E PESQUISAS EDUCACIONAIS ANÍSIO TEIXEIRA (INEP). **Censo Escolar, 2021**. Brasília: MEC, 2022. Disponível em: https://download.inep.gov.br/censo_escolar/resultados/2021/apresentacao_coletiva.pdf. Acesso em: 03 julho 2023.

ANTONIO e LEMOS. **As competências do professor do século XXI: a tecnologia como instrumento de apoio**. 2015. Disponível em: <https://periodicos.unisanta.br/index.php/hum/article/view/333/365>. Acesso em: 03 de julho de 2023.

GARLET, et al. **Uma proposta para o Ensino de Programação de Computadores na Educação Básica**. 2016. Disponível em: https://repositorio.ufsm.br/bitstream/handle/1/12961/TCCG_SIFW_2016_GARLET_DANIE LA.pdf?sequence=1&isAllowed=y. Acesso em: 03 de julho de 2023.

COMPUTAÇÃO NA ESCOLA. **Computação na Escola**, 2022. Disponível em: <https://computacaonaescola.ufsc.br/quem-somos/>. Acesso em: 03 de julho de 2023.

LYE, S. Y., KOH, J. H. L. **Review on teaching and learning of computational thinking through programming: Whatisnext for K-12?**. Computers in HumanBehavior, v. 41(C), p. 51-61. 2014. Disponível em: <<https://www.semanticscholar.org/paper/Review-on-teaching-and-learning-of-computational-is-Lye-Koh/64b5f719a6f7bff3c58e620d859d7dd5a3d3fdc1>>. Acesso em: 04 julho 2023

GONÇALVES, Pablo. **O que é programação em blocos?**, 2022. Disponível em: <<http://professorpablo.com/o-que-e-programacao-em-blocos/>>. Acesso em: 04 julho 2023

SCHILLER, J. et al. **Live Programming of Mobile Apps in App Inventor**. Proceedings of the 2nd Workshop on Programming for Mobile & Touch - PROMOTO '14, 2014. Disponível em: <https://dl.acm.org/doi/pdf/10.1145/2688471.2688482>. Acesso em: 04 julho 2023

FRANÇA, Gustavo. **Ambiente Web integrado com App Inventor para execução de aplicações Android**. 2019. TCC (Graduação) - Curso de Sistema da Informação, Universidade Federal de Santa Catarina, Florianópolis, 2019. Disponível em: <<https://repositorio.ufsc.br/bitstream/handle/123456789/202667/TCC.pdf?sequence=1&isAllowed=y>>. Acesso em: 04 julho 2023

MENDONÇA, R. A. R. de. **Levantamento de requisitos no desenvolvimento ágil de software**. Semana da Ciência e Tecnologia da PUC Goiás, p. 12, 2014.

SANJUGTHA, Shoba. **The Art of Exploratory Testing**. StickyMinds, 2020. Disponível em: <<https://www.stickyminds.com/article/art-exploratory-testing.>>. Acesso em: 05 julho 2023

Apêndice III - Sprints

A implementação do sistema *web* foi estruturada em *Sprints* de duas semanas, seguindo a abordagem ágil escolhida, com o objetivo de promover um desenvolvimento incremental e iterativo. Essa abordagem permite uma entrega contínua de funcionalidades ao longo do projeto, garantindo maior flexibilidade e adaptabilidade às necessidades em constante evolução.

Ao final de cada *sprint*, foram conduzidas reuniões de revisão, onde os resultados alcançados foram avaliados e melhorias foram identificadas. Essas sessões forneceram a oportunidade de refinar a abordagem e planejar adequadamente as próximas etapas do projeto.

A tabela abaixo a seguir apresenta um resumo das *sprints* realizadas, as escolhas dos requisitos se deu por dependência e grau de importância, assim destacando os componentes que receberam maior foco em cada uma:

Tabela 1: Tabela de apresentação das sprints

Requisito Funcional	Sprint								
	0	1	2	3	4	5	6	7	8
Setup Inicial	✓								
RF1 - Desenvolver o componente <i>ListView</i>		✓							
RF2 - Desenvolver o component <i>Notifier</i>		✓	✓						
RF3 - Desenvolver o componente <i>WebView</i>			✓						
RF5 - Desenvolver o componente <i>Camera</i>			✓	✓					
RF7 - Desenvolver o componente <i>Image</i>				✓					
RF9 - Desenvolver o componente <i>VideoPlayer</i>					✓				
RF4 - Desenvolver o componente <i>Camcorder</i>					✓				

RF9 - Desenvolver o componente <i>Player</i>						✓			
RF10 - Desenvolver o componente <i>Sound</i>						✓			
RF6 - Desenvolver o componente <i>ImagePicker</i>						✓			
RF13 - Alterar visibilidade dos componentes							✓		
RF14 - Refatoração variáveis globais							✓		
RF11 - Desenvolver o componente <i>TinyDB</i>							✓	✓	
RF12 - Refatoração das condicionais								✓	✓

Sprint 0

A *sprint 0* foi reservada para a preparação do ambiente de desenvolvimento, com o objetivo de estabelecer as bases necessárias para a implementação do projeto. Essa etapa envolveu a instalação da versão adequada do Node.js, bem como a configuração do gerenciador de pacotes Yarn. Essas medidas foram tomadas para garantir o funcionamento adequado dos pacotes já utilizados até então, como a biblioteca visual *Material UI*, que exigia uma versão específica.

A versão selecionada do Node.js para o ambiente de desenvolvimento foi a v16.17.1, com base em critérios técnicos e requisitos de compatibilidade com as dependências e bibliotecas do projeto. Essa versão específica foi considerada apropriada para garantir a estabilidade e o suporte necessário durante a implementação. Além disso, o gerenciador de pacotes Yarn foi instalado para facilitar o controle e a gestão eficiente das dependências do projeto (Figura 10). A versão escolhida foi a 1.22.19, que oferece recursos relevantes para a manutenção da consistência e da confiabilidade das dependências, incluindo a biblioteca *Material UI*.

```
C:\Users\Usuario>node --version
v16.17.1

C:\Users\Usuario>

C:\Users\Usuario>yarn --version
1.22.19
```

Figura 10: Versões do ambiente de desenvolvimento

Durante a *sprint* 0, também foram realizados testes para verificar a funcionalidade da conexão com o servidor Rendezvous. No entanto, durante esses testes, foram identificadas falhas na conexão, o que gerou a necessidade de investigar e compreender a natureza do problema, visando encontrar uma solução adequada.

Por meio da aplicação da lógica de engenharia reversa proposta por Darlan (2022) e da análise das informações contidas nas respostas dos pedidos realizados na aba de rede do navegador, foi possível identificar que a equipe do MIT havia realizado uma atualização nos parâmetros enviados ao servidor Rendezvous. Essa atualização teve impacto no *backend* desenvolvido na plataforma LIDEA-W, resultando em falhas na conexão. Após a identificação desse problema, foi tomada a decisão de atualizar a conexão, a fim de atender aos novos parâmetros definidos pela equipe do MIT. Essa atualização foi feita para garantir a compatibilidade correta entre o backend e o servidor Rendezvous.

Com a atualização da conexão conforme os novos parâmetros, foi possível restaurar o funcionamento adequado da conexão com o servidor Rendezvous. Essa solução permitiu restabelecer a comunicação entre as duas partes e garantir a continuidade do projeto de forma eficiente.

Sprint 1

A primeira iteração do projeto concentrou-se no desenvolvimento de dois componentes visuais principais: *ListView* e *Notifier*. Cada um desses componentes apresentou desafios distintos durante o processo de implementação.

No caso do componente *Notifier*, o desafio inicial estava relacionado à necessidade de acionamento por meio de uma função chamada por outro componente, como o *Button*. Devido a essa dependência, essa função precisou ser acessível fora do escopo do componente *Notifier*. A arquitetura existente do projeto não permitia o encadeamento de funções associadas a outros componentes, pois os componentes estavam isolados. Para acessá-los, eram utilizadas referências de identificadores únicos por meio do *Document Object Model* (DOM).

Considerando que o projeto utiliza a biblioteca *Material UI* para definir a aparência visual dos componentes, foi decidido utilizar o componente chamado *Dialogs*, que consiste em um tipo de janela modal que aparece à frente do conteúdo do aplicativo para fornecer informações importantes ou solicitar uma decisão. No entanto, esse componente utiliza um conceito denominado Portal, que permite renderizar os elementos em um nó DOM que existe fora da hierarquia DOM do próprio componente. Essa abordagem tornou o componente inacessível de acordo com a arquitetura inicialmente planejada para o projeto.

Para contornar esse problema foi utilizado *Context API* que é uma funcionalidade oferecida pelo React. Em vez de utilizar a propagação de propriedades através de múltiplos componentes, a *Context API* cria um contexto, que é um objeto que contém os dados que serão compartilhados. Esse contexto pode ser acessado por qualquer componente que esteja dentro do escopo desse contexto.

Por sua vez o componente *ListView* não dependia de acionamento via componente externo, entretanto deveria trabalhar modificando o valor de algum outro componente ou variável dentro da aplicação, caso contrário, não teria sentido - pois caso assim fosse, só seria uma lista que quando clicada nada acontece.

Logo, nessa primeira *sprint*, esse componente foi criado com a capacidade de modificar o texto de um componente *Label* do *App Inventor*, pois era a única modificação possível a ser realizada com o que estava implementado no projeto até então. Também foi realizada a função de busca dentro do componente de lista, essa funcionalidade podia ser habilitada ou desabilitada ao ajustar o componente dentro da configuração do aplicativo.

O maior desafio para esse componente foi entender o algoritmo da interpretação dos comandos Yail, como eles eram interpretados no código e a sequência dos métodos que eram executados quando na interface um elemento da lista era clicado. Então, foi necessário debugar o código até chegar na função responsável por alterar o conteúdo de texto de um componente *Label* e realizar as mudanças necessárias.

Sprint 2

Na segunda iteração do projeto, houve uma atenção especial dedicada ao desenvolvimento de três componentes: *WebViewer*, *Camcoder* e Câmera, sendo que *WebViewer* não possuía dependências em relação aos outros componentes. No entanto, durante essa etapa, surgiu um desafio inesperado, pois a funcionalidade do componente de imagem se tornou uma dependência entre os demais componentes, embora não tenha sido inicialmente identificada como um requisito funcional.

Diante dessa situação, uma adaptação foi realizada na *sprint*, aproveitando a flexibilidade da metodologia ágil adotada no projeto. O foco da *sprint* foi redirecionado para o componente *WebViewer*, que não possui dependência com o componente de imagem. Além disso, um novo requisito foi adicionado ao escopo e incluído na atual *sprint*, enquanto os outros dois componentes foram movidos para a próxima *sprint* para que a dependência pudesse ser resolvida de forma adequada.

A implementação do *WebViewer* utilizou o elemento HTML chamado *iframe*, que possibilita o carregamento de outra página HTML dentro do DOM. De forma essencial, o elemento *iframe* permite a inclusão de uma página da web dentro da página principal. A implementação do *WebViewer* foi realizada de maneira simples, sem apresentar nenhum problema significativo.

No entanto, para a implementação do componente de imagem, foi necessário um estudo e uma investigação mais aprofundados, que demandaram mais tempo do que o previsto inicialmente na segunda *sprint*. Como resultado, esse trabalho estendeu-se para a próxima *sprint* a fim de garantir uma implementação adequada.

Sprint 3

Considerando a implementação do componente de imagem como um aspecto de alta relevância, foi estabelecida uma decisão estratégica para direcionar o foco principal da terceira *sprint* do projeto para sua implementação. Essa determinação foi fundamentada em uma abordagem de engenharia reversa, iniciada na *sprint* anterior, com o objetivo de compreender minuciosamente o processo de envio de arquivos do servidor para o *AI2 Companion*.

Nessa etapa de estudo mais aprofundado, foi identificada a presença de um método HTTP específico, denominado *PUT*, que desempenha um papel fundamental no contexto do envio de arquivos binários. O método *PUT* é uma operação do protocolo HTTP que permite a criação ou substituição de recursos em um servidor, por meio do envio de dados.

No âmbito do *App Inventor*, esse requisição *PUT* é utilizado para enviar as imagens inseridas pelos usuários para o servidor. Essas imagens são tratadas como "*asset*" no contexto do *App Inventor*, representando recursos essenciais para o pleno funcionamento do aplicativo.

Durante a análise detalhada do código-fonte do *App Inventor*, foi possível identificar a implementação específica dessa requisição *PUT* e compreender como as imagens eram manipuladas nesse contexto. A compreensão minuciosa dos mecanismos e protocolos envolvidos no processo de envio de arquivos foi de suma importância para assegurar a correta implementação do componente de imagem no escopo do projeto.

Importante ressaltar que o fragmento de código apresentado na Figura 11, obtido por meio da análise do código-fonte do *App Inventor*, desempenhou um papel fundamental ao fornecer insights sobre o funcionamento da requisição *PUT* e sua interação com o envio e tratamento das imagens. Essa investigação permitiu uma visão mais precisa dos aspectos técnicos envolvidos, norteando as etapas subsequentes da implementação.

```

var conn = goog.net.XmlHttp();
var arraybuf = new ArrayBuffer(blob.length);
var arrayview = new Uint8Array(arraybuf);
for (var i = 0; i < blob.length; i++) {
  arrayview[i] = blob[i];
}
var rs = top.ReplState;
var encoder = new goog.Uri.QueryData();
//var z = filename.split('/'); // Remove any directory components
//encoder.add('filename', z[z.length-1]); // remove directory structure
var z = filename.slice(filename.indexOf('/') + 1, filename.length); // remove the asset di
encoder.add('filename', z); // keep directory structure

if (rs.proxy) {
  rs.proxy.postMessage(['asset', encoder.toString(), arraybuf], rs.proxy_origin);
  success(); // What happens if we fail?
} else {
  var conn = goog.net.XmlHttp();
  conn.retries = 3;
  conn.open('PUT', rs.baseurl + '?' + encoder.toString(), true);
  conn.onreadystatechange = function () {
    if (this.readyState == 4 && this.status == 200) {
      if (success) { // process callbacks
        success();
      }
    } else if (this.readyState == 4) {
      if (this.retries > 0) {
        this.retries--;
        this.open('PUT', rs.baseurl + '?' + encoder.toString(), true);
        this.send(arraybuf);
      }
    }
  }
}

```

Figura 11: Trecho do código fonte do método *putAsset* do *App Inventor*

Com base na análise realizada, foi implementada uma rota no *backend* do projeto para permitir que o servidor envie os arquivos para o *backend* da aplicação LIDEA-W. Essa rota é responsável por tratar as requisições *PUT* enviadas pelo *App Inventor* e processar os arquivos de imagem recebidos.

```

router.put('/', async (req, res, next) => {
  req.on('data', async (data) => {
    const pathImg = path.resolve(
      __dirname,
      '..',
      'uploads',
      req.query.filename
    );
    fs.appendFileSync(pathImg, Buffer.from(data), 'binary');
  });

  medias.add(req.query.filename);

  res.send({
    status: 'MEDIA',
  });
});

```

Figura 12: Trecho do código fonte do método *put* da plataforma LIDEA-W

O código apresentado na Figura 12 é responsável por tratar a requisição *PUT* enviada para o servidor LIDEA-W. Ao receber essa requisição, o código executa uma série de instruções para processar e armazenar a imagem enviada. Dentro dessa função, o código obtém o caminho do arquivo de imagem onde ele será armazenado. Esse caminho é determinado com base no nome do arquivo, que é obtido a partir do parâmetro *filename* presente na *query* da requisição.

Em seguida, o código utiliza a biblioteca *fs* para gravar os dados da imagem no arquivo especificado pelo caminho *pathImg*. A função *fs.appendFileSync* é usada para adicionar os dados da imagem ao arquivo, sendo que *Buffer.from(data)* converte os dados recebidos em formato binário.

Dessa forma, foi possível armazenar as imagens, assim como outros tipos de mídia que estavam anexadas no projeto dentro do *App Inventor* no *backend*. Entretanto, vale salientar que essa requisição responsável por enviar os dados binários das mídias, só é executada pelo *App Inventor* apenas uma vez, no momento em que se conecta ao *AI Companion*, ou seja, caso novas mídias forem adicionadas ao projeto, será necessário desconectar e conectar novamente para que o upload seja realizado.

Os *uploads* foram armazenados em uma rota estática dentro do *backend*, o que permite que eles sejam acessados externamente por outras aplicações, ou ainda ao digitar o caminho relativo da mídia desejada no navegador. Com isso um arquivo de imagem, chamado *imagem1.png* que está na pasta *uploads* dentro do *backend*, pode ser acessado ao se digitar no navegador: <http://enderecobackend/uploads/imagem1.png> e foi justamente esse endereço http o enviado ao *frontend*, junto com as informações do código YAIL que já eram enviadas

anteriormente. Assim, o componente imagem, que até então tinha apenas uma imagem fixa padrão, pode ser refatorado, recebendo valores dinâmicos.

Consequentemente, o componente *Camera* pode enfim ser implementado, uma vez que dependia do componente refatorado anteriormente. A ideia desse componente é acionar a webcam e possibilitar que o usuário tire uma foto e a veja na interface. Para realizar esse componente foi necessário recorrer a uma biblioteca externa chamada React-webcam. Essa biblioteca permite exibir no local desejado o conteúdo gerado pela webcam do usuário, assim como configurar as proporções exibidas na tela.

```

return (
  <div style={{ textAlign: "center" }}>
    <Webcam
      audio={false}
      ref={webCamRef}
      screenshotFormat="image/jpeg"
      videoConstraints={videoConstraints}
    />
    <button style={{ padding: "7px" }} onClick={capture}>
      Tirar Foto
    </button>
  </div>
);

```

Figura 13: Trecho do código fonte do componente Webcam da plataforma LIDEA-W

Na figura 13 é possível ver a estrutura básica do componente Webcam, ele possui um botão chamado “Tirar Foto”, que quando acionado pelo usuário é responsável por executar o método da biblioteca chamado *.getScreenShot()*, o qual é responsável por gravar o frame do momento em que foi clicado, gerando uma imagem de formato jpeg. Esse método retorna a imagem tirada em base64, ou seja, uma string que transmite dados binários em uma sequência de caracteres ASCII, a qual pode ser utilizada em um código html, dentro de uma tag src de um elemento imagem do HTML.

```

const capture = useCallback(() => {
  const imageSrc = webCamRef.current.getScreenshot();
  updateEvents([
    {
      type: "Image",
      componentName: componentName,
      method: "TakePicture",
      value: true,
      base64: imageSrc,
    },
    {
      type: "Camera",
      componentName: componentName,
      method: "TakePicture",
      value: false,
    },
  ]);
}, [webCamRef, updateEvents, componentName]);

```

Figura 14: Trecho do código fonte da função de *capture*

Seguindo a mesma ideia do componente *Notifier*, foi necessário recorrer ao *Context API* do React, para isso, foi criado um contexto de eventos geral dentro da aplicação, que seria utilizado quando um método de um componente fosse acionado e realizou alguma ação que resultasse em alterações em outro componente ou na interface. Com isso, foi possível transmitir a string em base64 para o componente Imagem, assim exibindo a foto recém tirada pela webcam na tela.

Sprint 4

Nesta quarta *sprint* foram desenvolvidos os componentes *VideoPlayer*, responsável por exibir na tela um vídeo e o componente *Camcorder*, com a função de gravar um vídeo da webcam do usuário e posteriormente exibi-lo na tela. Para se comunicar esses componentes necessitam utilizar a *Context Api* de Eventos criada na sprint anterior.

Primeiramente foi necessário focar no componente *VideoPlayer*, pois ele é um pré-requisito para o componente *Camcorder* poder exibir o seu vídeo recém gravado. Esse foi um componente teoricamente simples quanto a sua construção, uma vez que utilizou apenas o elemento `<video>` disponível no HTML, mas teve sua dificuldade na hora de receber os eventos, ou seja, na hora de interpretar o *play*, *pause*, *stop* executado dentro de um botão externo, além de receber o vídeo em formato binário do componente *Camcorder* e exibi-lo na tela. Na figura 15 é possível observar a estrutura básica do componente *VideoPlayer*.

```

{videoPath?.url} && (
  <div>
    <video
      id="videoControl"
      controls
      src={videoPath?.url}
      width={width}
      height={height}
      style={{ display: visible }}
      muted
    ></video>
  </div>
)

```

Figura 15: Trecho do código do componente *VideoPlayer*

Em seguida foi desenvolvido o componente *Camcorder*, que utilizou a biblioteca externa chamada *React-webcam*, a mesma utilizada no componente de *Camera*, mas agora utilizando a função de capturar um vídeo da webcam em funcionamento. Nele foi necessário utilizar os métodos específicos da biblioteca, a fim de criar um objeto que contém dados do tipo *MediaRecorder*, o qual é uma interface do Javascript responsável por gravar áudio e vídeo a partir de diferentes fontes.

Após essa etapa, foi necessário criar um objeto do tipo *Blob* utilizando os dados da captura do vídeo. *Blob* é usado para representar dados brutos, como um arquivo ou sequência de *bytes*, assim podendo ser utilizado para representar o vídeo recém gravado pela webcam. Na figura 16 é possível observar como foi criado o objeto do tipo *Blob*, dentro do componente.

```

const generateBlob = useCallback(() => {
  if (recordedChunks.length) {
    const blob = new Blob(recordedChunks, {
      type: "video/webm",
    });
    updateEvents([
      {
        type: "Video",
        componentName: componentName,
        method: "RecordVideo",
        value: true,
        video: blob,
      },
      {
        type: "Camcorder",
        componentName: componentName,
        method: "RecordVideo",
        value: false,
      },
    ]);
    setRecordedChunks([]);
  }
}, [recordedChunks, updateEvents, componentName]);

```

Figura 16: Trecho do código da função *generateBlob*

Esse arquivo *blob* foi enviado ao componente *VideoPlayer* via o contexto de eventos e então convertido para uma URL pelo método `URL.createObjectURL()`, assim sendo capaz de ser utilizado na tag `src` do elemento `<video>` disponível no HTML e com isso ser exibido na tela.

Sprint 5

Na *sprint 5* foi desenvolvido o componente *Player*, *Sound* e *ImagePicker*. O componente *Player* foi implementado para permitir a reprodução de arquivos de áudio, enquanto o *Sound* foi desenvolvido para controlar e ajustar os parâmetros do som. O *ImagePicker*, por sua vez, possibilitou a seleção e o carregamento de imagens no aplicativo.

Os componentes *Player* e *Sound* são componentes não visuais no *App Inventor*, ou seja, são elementos que não aparecem na tela do app, logo essa característica foi mantida nessa implementação. Ambos componentes possuem métodos responsáveis por fazer o áudio/som iniciar, pausar e parar e voltar ao início.

A implementação de ambos foi idêntica e de baixa complexidade, uma vez que ambos utilizaram o elemento `<audio>` do HTML para conseguir fazer a mídia tocar. Na figura 17 é

possível observar a estrutura simples do componente e a obtenção da url da mídia de som, que é transmitida através da *Context API - useContext(ContentContext)*.

```

export default function Sound({ componentProperties,
componentName }) {
  const { media } = useContext(ContentContext);

  const soundPath = media.find((media) => {
    return (
      media.name ===
      componentProperties.find((prop) => prop.propertyName ===
"Source")
        ?.propertyValue
    );
  });

  return (
    <audio
      controls
      id={componentName}
      src={soundPath?.url}
      style={{ visibility: "hidden" }}
    ></audio>
  </>
  );
}

```

Figura 17: Código do componente *Sound*

A diferença desses componentes ficou a cargo dos métodos dos mesmos. Enquanto o componente *Sound* possui os métodos: *Play*, *Pause*, *Stop* e o componente *Player* possui: *Start*, *Pause*, *Stop*. Na figura 18 é possível observar o bloco de código responsável por interpretar o comando vindo do componente *Player*. O algoritmo é responsável por decifrar o método acionado e então enviar informações para que ele seja executado. Como exemplo é possível observar quando o método encontrado é igual a “*Start*”, o algoritmo retorna uma instrução para que o componente *Player*, representado pelo elemento `<audio>` do HTML execute a função nativa `.play()`.

```

// Player
if (commands.includes("Player")) {
  const regexComponent = /call-component-method\s+'([\^']*)/;
  const matchComponent = commands.match(regexComponent);
  const componentName = matchComponent[1].trim();

  const methodValue = commands.split(`call-component-method `)[1];
  const regex = /'([\^']*).*\(\s*/;
  const match = methodValue.match(regex);
  const methodName = match[1].trim();

  const sound = window.document.getElementById(componentName);
  let action = "";
  if (sound) {
    if (methodName === "Start") {
      action = `document.getElementById("${componentName}").play()`;
    }
    if (methodName === "Pause") {
      action = `document.getElementById("${componentName}").pause()`;
    }
    if (methodName === "Stop") {
      action = `document.getElementById("${componentName}").pause();
      document.getElementById("${componentName}").currentTime = 0`;
    }
  }
  return `${action}`;
}

```

Figura 18: Trecho de código da tradução do YAIL para funções do componente *Sound*

O componente *ImagePicker* é um elemento especializado que atua como um botão com uma finalidade específica dentro de um sistema. Ao ser acionado pelo usuário, o *ImagePicker* invoca a exibição da galeria de imagens do dispositivo, oferecendo ao usuário a oportunidade de selecionar uma imagem de sua escolha. Após a seleção da imagem, esta é armazenada e a propriedade "*Selection*" assume o nome do arquivo no qual a imagem foi armazenada

Quanto à implementação em si, a complexidade foi relativamente baixa. O componente utiliza o recurso de botão disponibilizado pela biblioteca de interface *Material UI*, combinado com o elemento HTML `<input>` do tipo "file". O trecho de código abaixo ilustra essa implementação:

```
const handleClick = useCallback(() => {
  inputRef.current.click();
}, []);

return (
  <div>
    <input
      type="file"
      id="file"
      ref={inputRef}
      style={{ display: "none" }}
      accept="image/*"
      onChange={handleImage}
    />
    <Button
      onClick={handleClick}
      id={componentName}
      variant="contained"
      className={classes.button}
    >
      {properties.text}
    </Button>
  </div>
);
```

Figura 19: Trecho de código do componente *Image Picker*

Dessa forma, ao clicar no botão "Selecionar Imagem", o comportamento padrão do elemento HTML `<input>` do tipo `"file"` é acionado, permitindo a escolha da imagem desejada pelo usuário, proporcionando uma funcionalidade simples, mas eficiente, para a escolha e armazenamento de imagens em um aplicativo.

O desafio enfrentado nesse componente consistiu em atualizar o estado das mídias de forma a adicionar a imagem selecionada e torná-la acessível para o componente de imagem. Como mencionado anteriormente, os componentes possuem funções e propriedades isoladas. A solução encontrada foi criar uma função dentro do Contexto de Conteúdo (*ContentContext*) responsável por atualizar o array de mídias.

```

const updateMedias = (mediaItem, componentName) => {
  setMedia((prevMedia) => [...prevMedia, mediaItem]);
  const componentToUpdate = components.find(
    (item) => item.componentName === componentName
  );
  if (!componentToUpdate) {
    return;
  }

  const updatedProperties = [
    ...componentToUpdate.componentProperties.filter(
      (item) => item.propertyName !== "Picture"
    ),
    {
      propertyName: "Picture",
      propertyValue: mediaItem.name,
      propertyType: "text",
    },
  ];

  setComponents((prevComponents) => [
    ...prevComponents.filter((item) => item.componentName !== componentName),
    {
      ...componentToUpdate,
      componentProperties: updatedProperties,
    },
  ]);
};

```

Figura 20: Trecho de código da função *updateMedias*

Essa função, *updateMedias* (figura 20), é responsável por adicionar a *mediaItem* ao array de mídias através da função *setMedia*. Em seguida, localiza-se o componente a ser atualizado dentro do *array* de *components*, utilizando o nome do componente (*componentName*). As propriedades do componente são atualizadas, removendo a propriedade "Picture" (imagem anterior) e adicionando a nova imagem selecionada. Por fim, o estado de *components* é atualizado com as propriedades atualizadas do componente correspondente.

Essa abordagem permitiu a atualização eficiente do estado das mídias e das propriedades do componente de imagem, solucionando o desafio de forma adequada.

Sprint 6

Na sexta *sprint*, foi necessário desenvolver requisitos que até então não estavam previstos no cronograma inicial do projeto. A ideia inicial consistia em desenvolver apenas

componentes visuais do *App Inventor*, entretanto ao longo do desenvolvimento da aplicação, foi notado que algumas situações básicas esperadas não haviam sido implementadas ou não estavam funcionando corretamente. Com isso foi necessário inserir no escopo do projeto os requisitos: Alterar visibilidade dos componentes e refatoração de variáveis globais.

Uma situação muito comum encontrada em aplicativos desenvolvidos em qualquer tipo de linguagem é a de aparecer/desaparecer objetos na interface de acordo com o algoritmo criado. Até então o LIDEA-W possuía apenas a capacidade de interpretar do *App Inventor* o estado inicial de um componente, ou seja, se estava visível ou invisível, entretanto não era possível alterar essa visibilidade, ao executar alguma ação como um *click* em um botão.

```
const getBlocksCommands = (commands, variables) => {
  const stack = [];
  let parsedCommand = "";
  for (let i = 0; i < commands.length; i++) {
    // Pegar variavel local
    if (commands.startsWith(localVariableIndicator, i)) {
      getLocalVariables(commands, i);
    }
    // Mudando propriedade do componente
    else if (
      commands.startsWith(setPropertyIndicator, i) &&
      propertiesInsideIf === false
    ) {
      parsedCommand = getPropertyCommands(commands, variables, i);
      stack.push(parsedCommand);
    }
  }
}
```

Figura 21: Trecho de código da função *getBlocksCommands*

No trecho de código da Figura 21 é possível observar o método *getBlocksCommands*, um dos principais da aplicação, responsável por decifrar qual tipo de comando deve ser executado. Nesse trecho é possível observar caso a instrução tenha algum indicador que seja de mudar a propriedade do componente o método *getPropertyCommands* será executado.

O algoritmo até então implementado conseguia detectar todas propriedades do componente, entretanto havia casos em que ele não possuía ação nenhuma, como era o caso da visibilidade, ou seja, ele já era capaz de detectar que o componente possuía uma propriedade chamada “*Visible*”, mas não fazia nada. Com isso foi adicionado uma nova condição dentro desse método conforme a Figura22, essa condição foi implementada retornando a execução de um método *updateProperties* capaz de alterar no *Context API* geral da aplicação as propriedades do componente.

```

case "Visible":
  const visibility = commands
    .substring(i)
    .split("Visible ")[1]
    .substring(0, 2);

  setProperty = `updateProperties('${componentAction}', 'Visible', '${visibility}')`;
  if (commands.includes("#t (begin)" || commands.includes("#f (begin")
    setProperty = "";
    break;
  default:
    break;
}

propertiesStack.push(setProperty);
finalResult = propertiesStack.join("");
return finalResult.toString();
};

```

Figura 22: Trecho de código da *switch case*

Conforme descrito anteriormente, todo comando a ser interpretado na aplicação passa pelo método *getBlocksCommands*, logo foi necessário criar uma nova condição nesse método para conseguir realizar o requisito - alterar uma variável global, pois, até então não existia essa funcionalidade. Com isso foi adicionado o trecho de código da Figura 23

```

// Alterar variável global
else if (commands.startsWith(setGlobalVariableValue, i)) {
  const alreadyUpdated = stack.find((item) =>
    item.includes("updateGlobalVar")
  );
  parsedCommand = setGlobalVariable(commands, variables);
  if (alreadyUpdated && alreadyUpdated.includes("updateGlobalVar")) break;
  stack.push(parsedCommand);
}
}

```

Figura 23: Trecho de código da alteração de variável global

Esse trecho checa se o comando contém caracteres que indicam a modificação de uma variável global. Ele faz algumas verificações para ver se a variável já foi modificada e se inclui comandos como o *updateGlobalVar*, também executa o método *setGlobalVariable* que de fato retorna o valor da nova variável global, assim possibilitando que ela seja alterada no contexto geral da aplicação.

Na Figura 24 é possível observar o método *updateGlobalVar* e *updateProperties* que foram implementados dentro dessa *sprint* no arquivo de contexto geral de dados da aplicação LIDEA-W. Eles são os responsáveis por executar a atualização das variáveis globais e das propriedades, respectivamente.

```

const updateGlobalVar = (variable, newValue) => {
  const update = [...blocks];
  update.forEach((block) => {
    block.variables.forEach((v) => {
      if (v.variableName === variable) {
        v.variableValue = newValue;
      }
    });
  });
  setBlocks(update);
};

const updateProperties = (componentName, property, newValue) => {
  const update = [...components];
  update.forEach((component) => {
    if (component.componentName === componentName) {
      component.componentProperties.forEach((prop) => {
        if (prop.propertyName === property) {
          prop.propertyValue = newValue;
        }
      });
    }
  });
  setComponents(update);
};

```

Figura 24: Trecho de código das funções *updateGlobalVar* e *updateProperties*

Para que a visibilidade dos componentes fosse de fato fosse reativa, foi preciso adicionar em cada componente um método para detectar o estado atual da sua visibilidade, no exemplo abaixo podemos ver o trecho adicionado para possibilitar que a visibilidade fosse detectada em um componente. Caso ele identifique que o valor da propriedade é `#f` (*false*) ele retornar `none` assim na *tag style* do HTML foi passado o comando `display: none`, que faz o componente sumir; caso o valor fosse diferente de `#f`, retorna `block` o que faz o componente aparecer.

```

const visible = useMemo(() => {
  const value = componentProperties.find(
    (prop) => prop.propertyName === "visible"
  ).propertyValue;
  if (value === "#f") {
    return "none";
  } else {
    return "block";
  }
});

```

Figura 25: Trecho de código das funções *visible*

Sprint 7

Durante a *sprint 7*, ocorreu a implementação do componente *TinyDB*, um componente não visual de grande importância no armazenamento de dados. No contexto do desenvolvimento de aplicativos com o *App Inventor*, é necessário ressaltar que os aplicativos são reiniciados a cada execução, o que implica que os valores das variáveis não são preservados entre sessões subsequentes. Nesse sentido, o *TinyDB* foi introduzido como uma solução de armazenamento persistente, permitindo a retenção e disponibilização dos dados em todas as execuções do aplicativo.

Para a implementação do *TinyDB*, utilizou-se o *IndexedDB*, um recurso nativo dos navegadores, em conjunto com a biblioteca IDB. O *IndexedDB* é uma API poderosa para o armazenamento de dados no navegador, enquanto a biblioteca IDB facilita a interação com essa API, simplificando o processo de manipulação dos dados.

```
const dataBase = await openDB(dbName, dbVersion, {
  upgrade(db) {
    if (!db.objectStoreNames.contains(storeName)) {
      const store = db.createObjectStore(storeName, {
        keyPath: "tag",
        autoIncrement: true,
      });
      store.createIndex("tag", "tag", { unique: true });
    }
  },
});

setDb(dataBase);
```

Figura 26: Trecho de código da inicialização do *indexedDB*

Essa abordagem possibilitou o desenvolvimento de um mecanismo eficiente e confiável de armazenamento persistente por meio do *TinyDB*. Ao utilizar o *IndexedDB* e a biblioteca IDB, criou-se um ambiente adequado para a retenção dos valores das variáveis entre as execuções do aplicativo, proporcionando uma experiência contínua e consistente aos usuários.

Adicionalmente, é relevante mencionar que os itens de dados armazenados no *TinyDB* consistem em *tags* e valores. Para armazenar um item de dados, é necessário especificar a *tag* sob a qual ele deve ser armazenado. Essa *tag* é um bloco de texto que fornece um nome

identificador aos dados. Posteriormente, é possível recuperar os dados que foram armazenados sob uma determinada *tag*, utilizando a mesma *tag* como referência. Essa abordagem possibilita um acesso eficiente e organizado aos dados armazenados, facilitando a recuperação de informações específicas quando necessário.

Para evitar problemas de acesso às funções existentes nesse componente, foi utilizada a abordagem já mencionada de *Context*. Dessa forma as funções passam a estar disponíveis para todos os componentes que necessitem realizar armazenamento de informações (Figura 27).

```
return (  
  <TinyDbContext.Provider value={{ storeValue, getValueTag, cleanTag }}>  
    {children}  
  </TinyDbContext.Provider>  
)  
);
```

Figura 27: Trecho de código do *Context* do *TinyDB*

Além disso, durante esse *sprint*, foi realizada uma refatoração das condicionais do código. Essa refatoração teve como objetivo aprimorar a estrutura das declarações condicionais, tornando-as mais eficientes e legíveis. Essas melhorias contribuíram para a otimização do código e aprimoramento da sua manutenção.

Sprint 8

Conforme citado anteriormente, na sétima *sprint* além da finalização do componente *TinyDB* foi iniciada a refatoração das condicionais do código. Pois além de buscar otimizar o código e aprimorar sua manutenção foi detectado que as condicionais até então implementadas só funcionavam para o exemplo proposto no trabalho anterior, que era uma calculadora, ou seja, as condicionais só funcionavam quando envolviam a alteração de um conteúdo de texto de uma *label*.

```

else if (commands.startsWith(conditionalIndicator, i)) {
  parsedCommand = getConditionalCommands(commands, variables, i);

  if (parsedCommand.includes(callComponentMethod)) {
    const regex = /\{\{^\}\}*/g;
    const matches = parsedCommand.match(regex);

    let condition = parsedCommand.split(" ")[1];
    let instruction = ["", ""]; // [0] representa if; [1] representa else

    matches.forEach((match, index) => {
      let execute = match;
      if (match.includes(callComponentMethod)) {
        execute = parsedCallComponentMethod(match);
        instruction[index] += `${execute};`;
      }
      if (match.includes(setGlobalVariableValue)) {
        execute = setGlobalVariable(match, variables);
        instruction[index] += `${execute};`;
      }
      if (match.includes(setPropertyIndicator)) {
        const property = match.split("set-and-coerce-property!")[1];
        const propertyFix = `set-and-coerce-property!${property}`;
        execute = getPropertyCommands(propertyFix, variables, i - 1);
        instruction[index] += `${execute};`;
      }
    });
  }
};

```

Figura 28: Trecho de código da alteração da *Label*

Na figura 28 pode-se observar que assim como a alteração de visibilidade e de variáveis globais citadas na sexta *sprint*, o método responsável por checar as condicionais dos comandos, passa pelo método *getBlocksCommands*. e nele existe uma condição responsável por detectar se existe comandos a serem executados dentro de um bloco de condição, em caso positivo, o método *getConditionalCommands* é executado e retorna as instruções a serem realizadas. A variável *parsedCommand* contém a instrução condicional a ser executada, mas logo após ela ser gerada foi necessário incluir uma verificação para verificar se esse bloco condicional possui uma ação de executar um método de algum componente, pois apenas com o método *getConditionalCommands* não era possível identificar o comando de execução de um método.

Essa alteração possibilitou que comandos de blocos no *App Inventor* como o da figura 29: “Se a variável global name tiver o valor Ok, execute o método Start do componente *Video Player*, caso não mude o valor da variável global para Ok e faça o botão 3 não ser visível”, fossem identificados e executados na aplicação.



Figura 29: Imagem do *block Button*

Outra alteração necessária no método *getBlocksCommands* para fazer com que ele fosse capaz de se adaptar às novas funcionalidades foi a inclusão dessas duas checagens, representadas pela imagem abaixo, antes do comando ser enviado a pilha de instruções a serem realizadas.

```

if (parsedCommand.includes("updateProperties"))
    parsedCommand = parsedCommand.replaceAll("'", '"');

if (parsedCommand.includes("updateGlobalVar"))
    parsedCommand = parsedCommand.replaceAll("`", '"');

```

Figura 30: Trecho da função *getBlocksCommands*

Essas condições checam se o bloco de comando possui propriedades que alteram o *Context API*, caso sim, é necessário alterar alguns pedaços dessa *string*, para que o código seja interpretado de maneira correta ao ser executado, essas alterações permitiram que o bloco representado pela imagem abaixo, pudesse ser interpretado e executado de maneira correta.

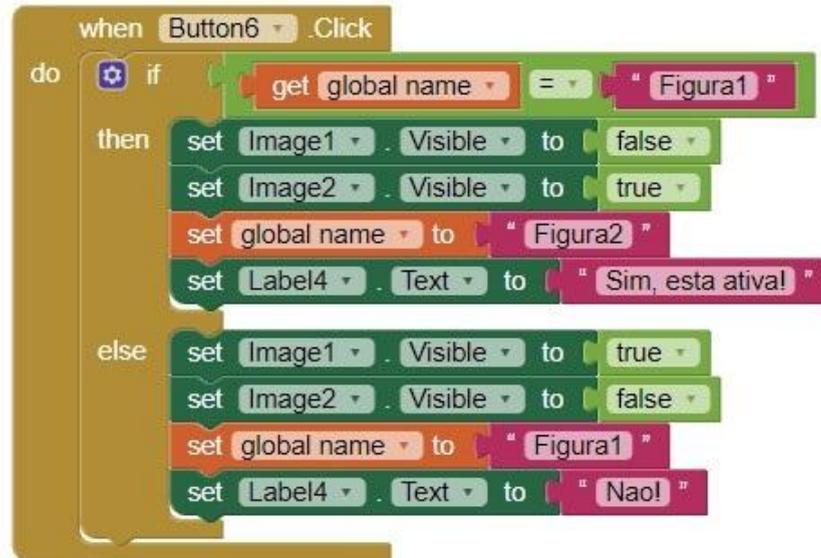


Figura 31: Imagem do *block Button6*

Entretanto, também foi necessário alterações no método *getConditionalCommands*, para que as condicionais pudessem ser otimizadas. Alterações como no código abaixo permitiram que as condições pudessem ser acumuladas, o que não estava acontecendo quando se tratava de ações diferentes de alterar o texto de uma *label*.

```

if (returnConditional.includes("if (false) {(any any)}")) {
  returnConditional = returnConditional.replaceAll(
    "if (false) {(any any)}",
    ""
  );

  returnConditional = returnConditional.replaceAll(` `);
  returnConditional = returnConditional.replaceAll(`);if`, `);if`);
}

return returnConditional;

```

Figura 32: Trecho da função *getConditionalCommands*

A inclusão do bloco condicional representado pela figura 32, antes do retorno da condicional, com a inclusão de algumas novas regras para interpretar a *string* de comando gerada, permitiu que um bloco de comandos do *App Inventor*, representado pela imagem abaixo, pudesse executar quantos ifs simples fossem necessários.

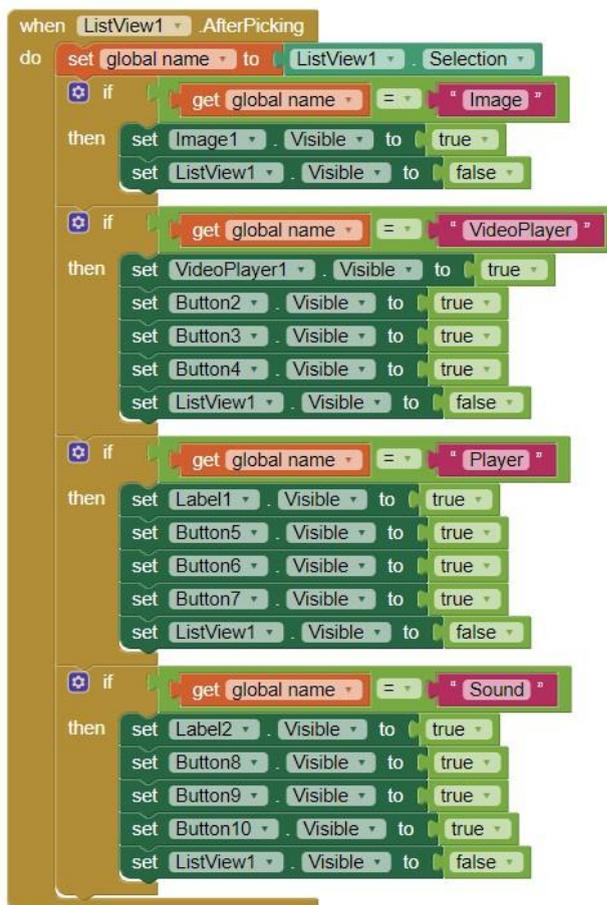


Figura 33: Imagem do *block ListView*

Com essas alterações a oitava *sprint* chegou ao final e as condicionais de interpretação do código YAIL da aplicação LIDEA-W foram aprimoradas, com isso, foi alcançado novas possibilidades de utilização de blocos condicionais de códigos feitos no *App Inventor*.