INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL-MECÂNICA
CURSO DE ENSINO SUPERIOR DE ENGENHARIA MECATRÔNICA

ÍSIS MACHADO SILVA

IMPLEMENTAÇÃO DE MELHORIAS NO PROCESSO DE DESENVOLVIMENTO COM A AUTOMATIZAÇÃO DOS TESTES DE INTEGRAÇÃO PARA APLICAÇÕES DESENVOLVIDAS À PLATAFORMA IOS

FLORIANÓPOLIS 2021

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA CÂMPUS FLORIANÓPOLIS
DEPARTAMENTO ACADÊMICO DE METAL-MECÂNICA
CURSO DE ENSINO SUPERIOR DE ENGENHARIA MECATRÔNICA

ÍSIS MACHADO SILVA

IMPLEMENTAÇÃO DE MELHORIAS NO PROCESSO DE DESENVOLVIMENTO COM A AUTOMATIZAÇÃO DOS TESTES DE INTEGRAÇÃO PARA APLICAÇÕES DESENVOLVIDAS À PLATAFORMA IOS

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia Mecatrônica pelo Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.
Orientador: Francisco Édson Nogueira de Melo

FLORIANÓPOLIS 2021

AGRADECIMENTOS

À minha mãe Simone e ao meu pai Jair, agradeço por todo amor, suporte e incentivo que me deram em toda minha jornada de vida até aqui.

Agradeço ao Quim por ser o melhor irmão que eu tenho e por ter me lembrado várias vezes, ao longo desse último ano, que eu precisava terminar meu TCC.

Às minhas amigas Lígia Rolim e Vittoria Palma sou grata por todo acolhimento, por sempre estarem me apoiando e incentivando a ser cada dia uma pessoa melhor.

À amiga Victória Votto sou muito grata por todo incentivo, apoio e por suas habilidades ortográficas. Sem ela, eu não sei o que seria deste trabalho.

Ao meu amigo Felipe Mesquita, agradeço por todo companheirismo durante essa incrível e longa jornada acadêmica.

Gostaria de agradecer ao Daniel Cordeiro Rossetti, à Nicole Dias e à Gabriela Campo Perez por sempre acreditarem no meu potencial e terem me incentivado a iniciar um estágio na Jungle Devs.

Agradeço ao Arthur Sady Cordeiro Rossetti por toda confiança depositada em mim, por todo incentivo, apoio e ajuda no meu desenvolvimento pessoal e profissional.

À empresa Jungle Devs agradeço por todo suporte, carinho e por toda ajuda para evoluir como desenvolvedora iOS e principalmente como pessoa.

Por fim, sou extremamente grata aos professores do curso de Engenharia Mecatrônica do IFSC por todo conhecimento compartilhado ao longo desses últimos anos.

MINISTÉRIO DA EDUCAÇÃO
SECRETARIA DE EDUCAÇÃO PROFISSIONAL E TECNOLÓGICA
INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS

DECLARAÇÃO DE FINALIZAÇÃO DE TRABALHO DE CURSO

Declaro que o estudante **Isis Machado Silva**, matrícula nº 131004314-0, do Curso de Engenharia Mecatrônica, defendeu o trabalho **Implementação de Melhorias no Processo de Desenvolvimento com a Automatização dos Testes de Integração para Aplicações Desenvolvidas na Plataforma IOS**, o qual está apto a fazer parte do banco de dados da Biblioteca Hercílio Luz do Instituto Federal de Santa Catarina, Campus Florianópolis.

Florianópolis, 16 de maio de 2021.

gov.br
Documento assinado digitalmente
Francisco Edson Nogueira de Melo
Data: 17/05/2021 16:29:27-0300
CPF: 261.107.173-04

Prof. Orientador do TCC: Francisco Édson N. de Melo

IMPLEMENTAÇÃO DE MELHORIAS NO PROCESSO DE DESENVOLVIMENTO COM A AUTOMATIZAÇÃO DOS TESTES DE INTEGRAÇÃO PARA APLICAÇÕES DESENVOLVIDAS À PLATAFORMA IOS

ÍSIS MACHADO SILVA

Este foi julgado adequado para obtenção do título de Engenheira em Engenharia Mecatrônica e aprovado na sua forma final pela banca examinadora do curso de Engenharia Mecatrônica do Instituto Federal de Educação, Ciência e Tecnologia de Santa Catarina.

Aprovada em: Florianópolis, 16 de maio de 2021.

Banca Examinadora:

Documento assinado digitalmente
gov.br
Francisco Edson Nogueira de Melo
Data: 17/05/2021 16:27:45-0300
CPF: 261.107.173-04

Prof. Me. Francisco Édson Nogueira de Melo

MAURICIO EDGAR STIVANELLO:90139062904
Assinado de forma digital por MAURICIO EDGAR STIVANELLO:90139062904
Dados: 2021.05.17 16:17:19 -03'00'

Prof. Dr. Mauricio Edgar Stivanello.

Documento assinado digitalmente
gov.br
Adriano Regis
Data: 18/05/2021 13:54:50-0300
CPF: 039.297.539-44

Prof. Me. Adriano Regis

DocuSigned by:
Arthur Sady Cordeiro Possetti
BFCA02FA587D4C7...

Eng. Arthur Sady Cordeiro Rossetti

RESUMO

No contexto de empresas que adotam o método Agile de desenvolvimento onde o cliente está presente no planejamento do projeto e demanda por entregas contínuas e um desenvolvimento colaborativo e flexível, fazem-se necessárias alternativas que possam otimizar o desenvolvimento do produto e a qualidade do mesmo.

A automatização de testes é um tema de grande importância quando se fala em qualidade de software pois possibilita que os desenvolvedores encontrem problemas mais rapidamente e com maior facilidade, fazendo com que menos problemas cheguem ao cliente ou ao usuário final, diminuindo a possibilidade de retrabalho e por conseguinte resultando em menos gastos com manutenção.

Existe uma grande variedade de testes automatizados, como por exemplo, teste unitário, teste de integração e teste de usabilidade. A aplicação destes envolve uma série de conceitos que nem sempre são triviais. Este trabalho irá focar nos testes de integração, responsáveis por testar componentes e testar a integração do código com ambientes externos, voltado a aplicativos iOS.

A partir dessa contextualização, o presente trabalho aborda o desenvolvimento de automatização de testes de integração aplicados à projetos iOS na Jungle Devs, empresa de desenvolvimento de software voltada a aplicações web e mobile. A princípio foi realizada uma pesquisa para analisar as ferramentas de testes de integração disponíveis no mercado, a partir da pesquisa realizada foi escolhida uma ferramenta de integração *open source* que atendia aos requisitos de teste e do projeto e por fim, realizou-se um comparativo da ferramenta escolhida com o *framework* nativo do XCode, ambiente de desenvolvimento para aplicações iOS.

Palavras chave: iOS, Testes automatizados, *software,* aplicativo.

ABSTRACT

In the context of companies that adopt the Agile method of development where the client is present in the project planning and demand for continuous deliveries and a collaborative and flexible development, alternatives are needed that can optimize the product development and the quality of the same.

Test automation is a topic of great importance when it comes to software quality because it allows developers to find problems faster and more easily, making fewer problems reach the customer or the end user, reducing the possibility of rework and therefore resulting in less maintenance expenses.

There is a wide variety of automated tests, such as unit testing, integration testing and usability testing. The application of these involves a series of concepts that are not always trivial. This work will focus on integration tests, responsible for testing components and testing the integration of code with external environments, aimed at ios applications.

From this context, the present work addresses the development of automation of integration tests applied to iOS projects at Jungle Devs, a software development company focused on web and mobile applications. At first, a research was carried out to analyze the integration testing tools available on the market. From the research carried out, an open source integration tool was chosen that met the test and project requirements. Finally, a comparison of the tool chosen with the native xcode framework, development environment for iOS applications.

Key words: *iOS, automated test*, *software, apps*.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

Por definição, o desenvolvimento de software é a ação de elaborar e implementar um sistema computacional, isto é, transformar a necessidade de um utilizador em um produto digital. É também compreendido como o processo de aplicação de conceitos da engenharia de software, combinados a uma pesquisa de mercado e necessidades de um utilizador que por finalidade resultará em um software. Sistemas de software atuais são cada vez maiores em tamanho, número de usuários e complexidade estrutural. Para que sejam dignos de confiança, tais sistemas devem ser capazes de prover seus serviços corretamente mesmo operando sob condições excepcionais. Mecanismos de tratamento de exceções (GOODENOUGH, 1975) proveem facilidades para a detecção e a sinalização de condições excepcionais. Tais mecanismos disponibilizam ainda funcionalidades para estruturar ações que remedeiem a ocorrência dessas condições excepcionais.

Entretanto, implementar o tratamento de exceções adequadamente não é uma tarefa fácil de ser realizada. Mesmo desenvolvedores experientes têm dificuldade em implementar código de tratamento de exceções de boa qualidade.

O processo de desenvolvimento de software normalmente é dividido em etapas, são elas: Análise da necessidade e especificação de requisitos, construção do produto, testes e correção de erros, falhas ou defeitos envolvidos na criação e manutenção do software e por fim, entrega.

É importante ressaltar que o modelo de processo de software (conjunto distinto de atividades, ações, tarefas e marcos que levam à produção de um produto de software) utilizado é, talvez, o grande e maior contribuinte para o sucesso ou fracasso de um projeto de software, pois este influencia diretamente no modo como serão conduzidos a comunicação, o planejamento, a modelagem, a construção e a implantação. Sendo assim é de extrema importância a escolha adequada do modelo a ser aplicado para o projeto de desenvolvimento de software em questão, pois cada projeto pode requerer um modelo diferente (PRESSMAN, 2006).

As fases/etapas de desenvolvimento podem ser abordadas de formas diferentes entre empresas ou entre projetos dentro de uma mesma empresa. O presente trabalho apresentará as etapas de desenvolvimento de software com base nos processos adotados pela Empresa Jungle Devs. Propondo uma melhoria no

próprio processo de desenvolvimento de software, implementando testes de integração automatizados entre as fases de construção do produto.

Na Empresa Jungle Devs, é adotado o modelo Agile de desenvolvimento. O modelo Agile tem como base o princípio de satisfazer o cliente através de entregas contínuas, adotando requisitos de desenvolvimento e soluções por meio do esforço colaborativo e organizado entre o desenvolvedor e o dono do produto, procurando, assim, manter uma comunicação contínua com o cliente e com os membros da equipe.

Este modelo baseia-se no planejamento adaptativo, constante evolução no desenvolvimento, entrega antecipada, melhoria contínua e incentiva respostas flexíveis à mudança.

Dentro do contexto da empresa Jungle Devs, em que o processo de desenvolvimento é adaptativo e mutável; medidas que possibilitam a agilidade de desenvolvimento e otimizar a qualidade do produto a ser desenvolvido são de extremo interesse para a empresa.

A automatização dos testes de integração é uma tecnologia relativamente nova no mercado e muitas empresas de software já aderiram aos testes de integração como parte do processo de desenvolvimento. Ela permite que o desenvolvedor encontre com maior facilidade e mais rapidamente os erros (famosos bugs) presentes no software.

O teste de software é basicamente uma técnica de programação cujo objetivo é testar e comparar o resultado obtido com o resultado esperado. Isso pode ser alcançado escrevendo funções responsáveis por testar partes do programa ou usando qualquer ferramenta de automação de testes de software. A automação de teste é usada para automatizar tarefas repetitivas e outras tarefas de teste que são difíceis e demandam muito tempo para executar manualmente.

No presente trabalho realizou-se um comparativo entre duas ferramentas de teste compatíveis com a linguagem Swift.

A linguagem Swift é a linguagem utilizada para desenvolver aplicações/programas para a plataforma iOS. A partir deste comparativo, foi possível eleger a ferramenta que melhor se adequa aos requisitos do projeto. O comparativo levou em consideração a velocidade com que os testes foram executados, percentual de cobertura de código, facilidade ao instalar a ferramenta e facilidade de uso da ferramenta.

1.1 Justificativa

Em um cenário de desenvolvimento de software em que a empresa ou o próprio desenvolvedor não se preocupam em testar ou integrar a disciplina de teste ao desenvolvimento, permite-se que o programa fique suscetível à situação de deixar o usuário final/cliente na posição de testador final do produto e por consequência, determinando o sucesso financeiro do aplicativo. Isso é ruim pois se o produto tiver algum erro ou não funcionar adequadamente, o usuário final ou o cliente pode perder credibilidade na empresa ou pessoa contratada para desenvolver o programa.

Empresas de sucesso já usufruem da disciplina de teste automatizados em seus produtos e aprimoram constantemente suas abordagens de desenvolvimento de software para ser o mais eficiente possível. Isso permite que as equipes de desenvolvimento dediquem seu tempo a inovações ao em vez de retrabalho e tarefas manuais sujeitas a erros. Um exemplo disso é a empresa Google que desenvolveu uma estrutura de teste automatizado próprio para testar os próprios aplicativos desenvolvidos pela empresa.

Ter a capacidade de obter uma avaliação sobre a qualidade do software ao longo de todo o processo de desenvolvimento do aplicativo auxilia os desenvolvedores e gestores a prever com maior precisão, eficiência e rapidez o estado do software, possibilitando um planejamento melhor para as próximas demandas de tarefas, além de poder colocar o software nas mãos do usuário final o mais cedo possível.

Em suma, o teste contínuo significa que as organizações não precisam esperar que todas as partes do aplicativo fiquem prontas antes que os testes possam começar. O teste contínuo é a verificação do software por meio de uma variedade de técnicas específicas, incluindo a execução de testes de unidade criados pelos programadores a cada função implementada.

Quanto mais cedo e com mais frequência uma organização pode obter avaliações sobre a qualidade de seus produtos, mais cedo ela pode reagir a falhas de arquitetura, decisões de *design* inadequadas, funcionalidade de aplicativo incorreta, vulnerabilidades de segurança e problemas de escalabilidade. O teste contínuo é a prática em que esse recurso pode ser uma realidade.

1.2. Definição do problema

Ao falarmos em desenvolvimento de soluções digitais como aplicativos, devemos levar em conta todos os processos envolvidos em sua produção até o resultado final. O processo de desenvolvimento é dividido em etapas que consistem basicamente em análise, programação, integração do código, testes e entrega.

Normalmente, no desenvolvimento de aplicativos o processo de desenvolvimento é dividido em dois principais segmentos de serviço, mais popularmente conhecidos como *frontend* e *backend* (LINDLEY, 2018). O desenvolvedor *frontend* trabalha com a parte visual da aplicação que interage diretamente com o usuário. Enquanto o desenvolvedor *backend* trabalha na parte de infraestrutura da solução ou não visual da aplicação. Este último é o responsável, em termos gerais, por determinar as regras de funcionamento da aplicação, além de implementar o armazenamento das informações que podem ser disponibilizadas pelo usuário e as informações a serem mostradas ao usuário.

Estas duas linhas de serviço são totalmente dependentes uma da outra. Ou seja, para que o *frontend* funcione, o *backend* deve antes estar configurado e funcionando de acordo com o previsto. Nessa “linha de produção” / “*pipeline*” de tarefas é evidenciada a defasagem no tempo disponível para que os times de diferentes linhas de serviço possam trabalhar. Isso é um problema pois permite que o time trabalhando no *frontend* fique impedido de trabalhar no projeto até que o time responsável pelo *backend* tenha terminado seu trabalho. Outro problema enfrentado na integração desses serviços acontece quando o backend altera a lógica de funcionamento ou até mesmo os parâmetros usados na aplicação. Isso pode resultar na quebra do aplicativo sem nenhuma influência da pessoa responsável por implementar a interface (frontend), caso não haja a devida comunicação entres os times de desenvolvimento envolvidos no projeto.

Além disso, um dos maiores desafios com a integração desses dois segmentos de trabalho é que os problemas apresentados no projeto normalmente permanecem ocultos durante todo o processo de desenvolvimento e aparecem apenas no final da implementação quando o produto é finalmente testado, levando a custos extras, atrasos, qualidade inferior do produto e, às vezes, até falha do projeto. Por esse motivo os testes de integração são de fundamental importância, pois o mesmo verifica

a integralidade entre os diferentes componentes da aplicação, alertando o desenvolvedor onde encontra-se o erro.

O processo de integração do código pode se dar de forma tradicional ou contínua. O modo tradicional pode apresentar um “ponto de dor” no desenvolvimento do produto pois consome tempo da equipe, desenvolvendo de forma dependente e principalmente testando manualmente o código. Além disso, caso o programa venha a apresentar defeitos, como falado anteriormente, serão verificados apenas ao final do processo, desencadeando uma necessidade de retrabalho. Isso implica na reunião do time para averiguar onde o erro se encontra no código, análise para determinar a pessoa responsável a consertar o erro e finalmente refazer todo o processo até que o programa responda como o esperado.

1.3. A empresa

A Jungle Devs é uma empresa de desenvolvimento de software focada no desenvolvimento de pessoas que trabalham ou pretendem trabalhar no setor de tecnologia da informação (TI). Sediada em Florianópolis, Santa Catarina, a empresa atua na área de desenvolvimento de software, com especialidade em produtos web e mobile, além de oferecer programas de capacitação profissional para pessoas que anseiam em atuar no setor de tecnologia.

A empresa foi fundada em fevereiro de 2018 por ex-alunos de Engenheira de Controle e Automação da UFSC. A ideia de fundar a empresa surgiu a partir da identificação da necessidade de profissionais atuantes no setor de tecnologia. A Jungle Devs surgiu então com a missão de repensar a forma como as pessoas aprendem e criam tecnologia, dando o início desse processo com um programa de capacitação denominado Academy. Os participantes do Academy passam por diferentes períodos de treinamento, partindo do aprendizado de conceitos básicos de programação até a atuação no desenvolvimento de um projeto real e conexão do profissional emergente com o mercado de trabalho.

2. OBJETIVOS

A seguir serão apresentados o objetivo geral e os objetivos específicos que serviram de base para o desenvolvimento do presente trabalho.

2.1. Objetivo geral

Este trabalho tem como principal objetivo avaliar a utilização de testes de integração como ferramenta para otimização do processo de desenvolvimento de projetos da empresa Jungle Devs por meio da implementação de testes contínuos de integração entre os segmentos de *frontend* e *backend,* propondo que os desenvolvedores responsáveis pela disciplina de *frontend* sejam capazes de testar imediatamente o código e averiguar com maior facilidade e rapidez a presença de erros/*bugs*.

O objetivo de adotar a integração contínua é estabelecer processos mais robustos que possam ser implementados em projetos de maior impacto, onde o custo de erros pode ser elevado.

2.2. Objetivos específicos

- Elaborar uma pesquisa com o intuito de descobrir como as tecnologias voltadas a testes automatizados estão sendo usadas nos processos de desenvolvimento de software atualmente.
- Analisar comparativamente as ferramentas disponíveis, de modo a verificar a capacidade de atenderem aos problemas vivenciados pela empresa Jungle Devs.
- Escolha da ferramenta que melhor atende os problemas de integração, comparar com o modelo usado anteriormente e avaliar se valeria ou não a pena o esforço de implementar testes contínuos nos projetos da empresa.

3. FUNDAMENTAÇÃO TEÓRICA

No presente capítulo, será apresentado alguns dos conceitos básicos necessários para melhor compreensão do trabalho desenvolvido.

3.1. *Software*

Software é, por definição, um conjunto de comandos ou programas que instruem um computador ou dispositivos como telefones celulares, tablets ou outros, a executar determinadas tarefas.

No século 19, a matemática e escritora Ada Lovelace, projetou e desenvolveu o que veio a ser o primeiro *software*. Ela criou um algoritmo para mostrar como a máquina analítica projetada por Charles Babbage poderia calcular a sequência de Bernoulli.

Apesar de Ada Lovelace ter criado o primeiro *software* no século 19, a teoria do *software* foi formulada pela primeira vez por Alan Turing em 1935, propondo que o *software* seria uma sequência de números computáveis com uma aplicação para o problema de decisão.

Softwares podem ser divididos em três principais categorias, as quais serão exploradas a seguir (ROSENCRANE, 2021).

3.1.1 *Software* de programação

São programas que auxiliam no desenvolvimento de outros programas, mais conhecidos como IDEs do inglês "*Integrated Development Environment*", os *softwares* de programação podem ser também: compiladores, vinculadores, depuradores. Um exemplo dessa categoria de *software é* o XCode, programa/ambiente de desenvolvimento utilizado neste projeto.

3.1.2 *Software* de sistema

O *software* de sistema é o *software* encarregado de fazer a comunicação entre o *hardware* e o usuário, e serve como base para o *software* aplicativo. O *software* do

sistema inclui drivers de dispositivo, sistemas operacionais (SO), compiladores, editores de texto e utilitários que ajudam o computador a operar com mais eficiência.

Ele também é responsável por gerenciar componentes de hardware. O *software* de sistema geralmente é escrito na linguagem de programação C, que é uma linguagem classificada como sendo de baixo nível, ou seja, mais próxima com a linguagem de máquina do que com a linguagem humana.

Os sistemas operacionais como Windows, macOS, Linux, iOS, Android, por exemplo, são *softwares* de sistema. O presente projeto foi desenvolvido no sistema operacional macOS (sistema operacional da Apple destinado a MacBooks e iMacs), propondo soluções aplicadas a programas desenvolvidos para a plataforma iOS.

3.1.3 *Software* aplicativo

O *software* aplicativo é responsável por executar um grupo de funções, tarefas ou atividades coordenadas para o benefício do usuário. Exemplos de *software* de aplicativo incluem suítes de escritório como o Microsoft Office, aplicativos de jogos, s*oftware* educacional, entre outros.

O presente projeto foi desenvolvido em cima do *software* aplicativo *The New Life,* desenvolvido pela empresa Jungle Devs.

3.2. *Open Source* (*Software* de Código Aberto):

Open Source ou *software* de código aberto é um tipo de *software* de computador em que o código-fonte é disponibilizado para a comunidade sob uma licença na qual o detentor dos direitos autorais concede aos usuários os direitos de usar, estudar, alterar e distribuir o *software* para qualquer pessoa e para qualquer finalidade.

O *software* de código aberto pode ser desenvolvido de forma pública e colaborativa (OPENSOURCE, 2007)

.

3.3. iOS

O iOS é o sistema operacional móvel criado e desenvolvido pela Apple Inc. e aplicado exclusivamente a alguns hardwares desenvolvidos pela mesma, como o iPhone, iPod Touch e o iPad. É também a base para outros sistemas operacionais criados pela empresa como o macOS, tvOS e watchOS.

Por ser um software para dispositivos móveis, ele é baseado no conceito de manipulação direta. Isso significa que a interação é feita a partir de toques na tela ou deslizar os dedos para realizar diferentes ações.

O sistema operacional iOS é o segundo sistema operacional móvel mais instalado do mundo e é baseado nas linguagens de programação C, C++, Objective-C, Swift e Java. Seus programas interpretam as linguagens Objective-C e Swift (BERTOLI, 2014).

3.4. Swift

O Swift é uma linguagem de programação criada pela Apple. Seu principal desenvolvedor foi o engenheiro de software Chris Lattner, que começou o projeto de desenvolvimento do Swift em 2010 e somente em 2 de junho de 2014 a linguagem foi lançada publicamente.

Antes do lançamento do Swift, os sistemas operacionais iOS e MacOS da Apple eram baseados na linguagem de programação Objective-C, uma linguagem orientada a objeto baseada em linguagem C.

O Swift é projetado para trabalhar com os principais *frameworks* desenvolvidos pela Apple, além de ser compatível com o Objective-C, havendo possibilidade de utilizar as duas linguagens em conjunto em um mesmo programa (SWIFT, 2021).

3.5. Gerenciamento de Dependências

CocoaPods é um gerenciador de dependência a nível de aplicativo para projetos desenvolvidos nas linguagens Objective-C e Swift, que fornece um formato padrão para gerenciar bibliotecas externas.

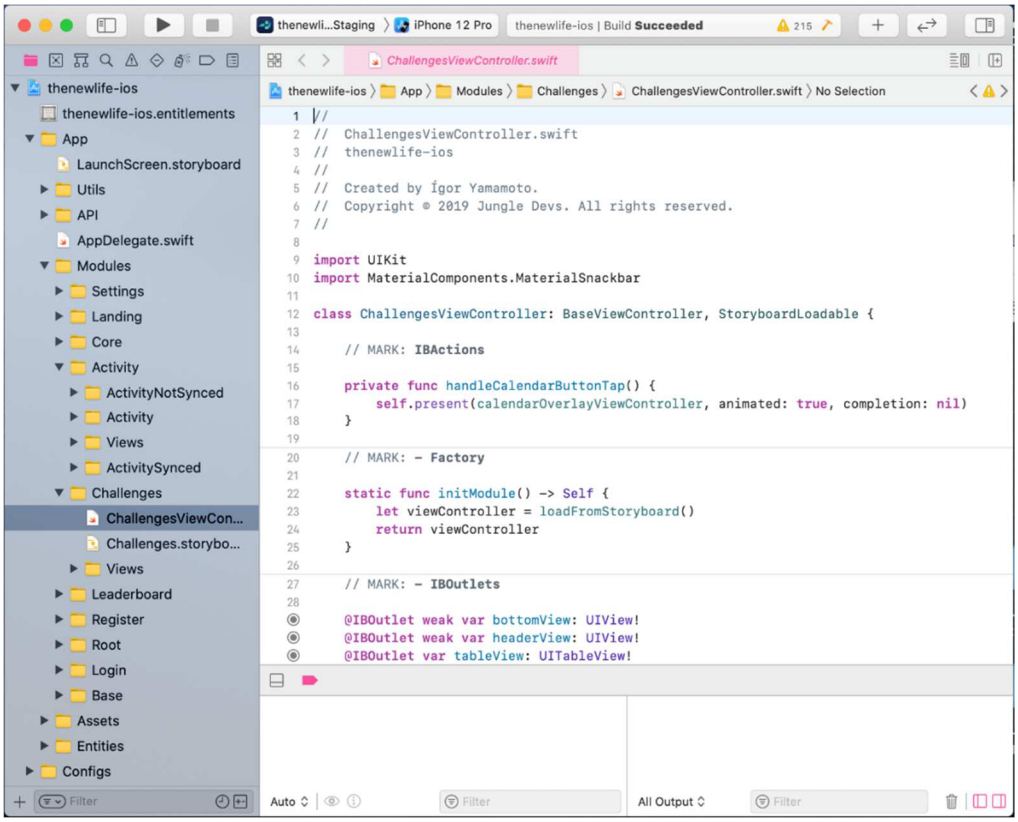
O CocoaPods foi pensado de forma a ser de fácil aplicação integrando-se automaticamente em projetos desenvolvidos com Xcode. Para instalar dependências

no projeto usando CocoaPods, basta navegar até a pasta do projeto pelo terminal, rodar o comando "pod init", será criado automaticamente um arquivo chamado "pod file" na pasta do projeto. Para instalar a dependência desejada, basta escrever o nome do arquivo no pod file e rodar a linha de comando "pod install" no terminal (COCOAPODS, 2021).

3.6. Xcode

O Xcode é o ambiente de desenvolvimento integrado (IDE) (figura 1) da Apple. A ferramenta possui um editor de texto integrado (figura 1), uma ferramenta de construção de interfaces (interface builder) e outras funcionalidades que dão suporte ao desenvolvimento de aplicativos para os sistemas iOS, macOS, watchOS e tvOS. O Xcode possui o SDK (Software Development Kit - Coleção de ferramentas de software que servem para facilitar o desenvolvimento de aplicativos) iOS integrado, o qual permite a compilação, a simulação de dispositivos, entre outros recursos (XCODE, 2021).

Figura 1 - Ambiente de desenvolvimento do XCode



Fonte: Autoria própria

3.7. Métodos Ágeis de desenvolvimento

Metodologia ágil é um modelo e uma filosofia que propõe alternativas à gestão de projetos tradicionais e tem a função de aprimorar o processo de desenvolvimento de um produto ou serviço. O objetivo final é fazer entregas com rapidez e com maior frequência, conforme surgem as necessidades do cliente.

3.7.1. Agile

Agile (desenvolvimento ágil de software ou métodos ágeis) é um termo que se refere a abordagem de desenvolvimento de soft*ware* em que requisitos e soluções evoluem através de um esforço colaborativo de times autogeridos e multi-funcionais, de seus clientes e usuários finais. O conceito prega planejamentos adaptativos, desenvolvimento iterativo com melhorias contínuas e encoraja a rápida e flexível resposta a mudanças. O conceito é frequentemente posto em prática com o uso de *frameworks* de desenvolvimento, como o Scrum (CRISPIN, 2021).

3.7.2. Scrum

Scrum é uma estrutura processual que tem como propósito abordar e resolver problemas mutáveis e adaptativos de forma clara, organizada e frequente. O f*ramework* fundamenta-se nos conceitos de transparência, inspeção e adaptação e tem como valores: compromisso, coragem, foco, abertura e respeito (SCHWABER, 2020).

O Scrum consiste em times com atribuições associadas, eventos e artefatos. Estes elementos são elencados a seguir:

Atribuições:

- Gerente de projeto: É a pessoa responsável por gerenciar o desenvolvimento do produto.
- Time de desenvolvimento: Estes devem ter seus respectivos trabalhos autogeridos de tal forma a aumentar a eficiência e eficácia do projeto.

Eventos:

- *Sprint*: é o evento central do Scrum, consistindo de uma janela limitada de tempo com duração de 15 dias. Durante este período temporal, a equipe deve concluir uma versão incremental do produto a ser entregue. As durações das sprints devem ser consistentes ao longo do projeto, com uma sprint iniciando-se imediatamente ao término da anterior.
- Planejamento da *Sprint*: o trabalho a ser realizado durante a sprint é planejado durante a sessão de planejamento da sprint através do trabalho colaborativo de todos os membros do time.
- *Daily Scrum*: é um evento diário executado pelo time de desenvolvimento, limitado a quinze minutos. Ocorre todos os dias no mesmo horário com o objetivo de inspecionar o trabalho executado no dia anterior e anteceder discussões sobre o trabalho futuro.
- Revisão da *Sprint*: evento que ocorre ao final da sprint com o intuito de avaliar o progresso do time em relação ao incremento do produto. O time discute o que foi feito e provê feedback.
- Retrospectiva da *Sprint*: evento que ocorre após a revisão da *sprin*t e antecede o planejamento da próxima *sprint*. Durante o evento, o time tem a oportunidade de avaliar seu próprio desempenho e criar um plano de melhorias a serem alcançadas na próxima *sprint*.

Artefatos:

- *Product Backlog*: consiste em uma lista ordenada de todas as tarefas a serem executadas que são de conhecimento até o momento. É uma única fonte de requisitos para qualquer mudança a ser implementada no produto. Ele nunca está completo, mudando constantemente conforme a evolução do produto. Ele lista todas as funcionalidades, requisitos, melhorias e consertos que devem ser aplicados ao produto, bem como uma descrição do que é considerada uma tarefa completa.
- *Sprint Backlog*: é um conjunto de itens selecionados do backlog do produto com um plano do incremento do produto que será entregue ao final da sprint.

3.8. *Frontend* e *Backend*

Por definição, os termos *frontend* e *backend* referem-se à separação de interesses entre a camada de apresentação (*frontend*) e a camada de acesso a dados (*backend*) de um pedaço do software.

O desenvolvedor *backend* trabalha na parte de infraestrutura da aplicação. Ele é o responsável, em termos gerais, pela implementação da regra de negócio.

Quando falamos de *backend*, nos deparamos com várias linguagens, como Java, Python, Ruby, entre outras. Cada uma possui vantagens e desvantagens em relação ao uso no desenvolvimento, bem como no mercado de trabalho.

O desenvolvedor *frontend* é responsável pela interface. Trabalha com a parte da aplicação que interage diretamente com o usuário. Por isso, é importante que esse desenvolvedor também se preocupe com a experiência do usuário (YAMAMOTO, 2018).

3.9. Testes de *Software*

Os testes de *software* fazem parte do processo de desenvolvimento de *software* e têm como função identificar, registrar e documentar erros estruturais ou de código, visando corrigi-los até que o software atinja a qualidade esperada.

A área de estudo de testes de software é vasta, contemplando diversos tipos de testes com diferentes propósitos e maneiras de serem implementados, como os que seguem (CUNHA, 2010):

- Teste de fumaça: checagem básica das funcionalidades do software;
- Teste de regressão: detecção de erros após mudanças no código;
- Teste de aceitação: verificação se a solução funciona para o usuário;
- Teste de usabilidade: verificação da facilidade de uso do sistema;
- Teste exploratório: uso do sistema de forma não-determinística.

3.10. Tipos de Testes de *Software*

Os tipos de testes são as formas de se testar uma aplicação, esses testes se diferem devido as técnicas empregadas para sua execução e não em seu ambiente. Nos atentaremos aqui aos quatro principais tipos de testes empregados atualmente. São esses (ALVES, 2018):

- Testes de Unidade/Testes Unitários: Esse tipo de teste visa a isolar uma pequena parcela específica do código e verificar se sua funcionalidade corresponde ao esperado.
- Teste de Integração: Possui foco em verificar a funcionalidade dos módulos individualmente assim como integrados com outros módulos da aplicação, em grupos, são responsáveis por testar a interação entre diferentes componentes em resposta a entradas no sistema. Geralmente dependem da conclusão do desenvolvimento da aplicação para sua execução.
- Testes de interface gráfica: Simulam o comportamento do usuário final do sistema frente a interface gráfica do software (YAMAMOTO, 2018).

3.11. Testes de *Software* Automatizados

Testes de *software* automatizados consistem basicamente em funções integradas ao próprio programa, que são responsáveis por testar outras funções ou partes do código. Para isso pode-se fazer uso de ferramentas externas para controlar a execução dos testes.

A automação permite que o teste seja repetido várias vezes, sendo mais fácil encontrar novos erros através da repetição e da simulação de cenários específicos.

O objetivo final dos testes automatizados é minimizar os problemas da abordagem manual, o tempo despendido, o retrabalho do desenvolvedor e, consequentemente, o custo final (CUNHA, 2010).

Existem vários tipos de testes automatizados, os principais são:

- Teste de usabilidade: Teste para encontrar disfuncionalidades no uso. Esse tipo de teste é fortemente utilizado para medir como será a interação do usuário com o software.
- Teste de segurança: Utilizado para encontrar brechas de segurança no software.
- Teste de regressão: Permite detectar erros após mudanças no código. Esse tipo de teste é mais facilmente mensurável, já que valida se o comportamento de uma (ou algumas) aplicações possuem o comportamento esperado.
- Teste de desempenho: Um teste de desempenho pode testar coisas como tempo de resposta do *backend* logo após uma interação do cliente, e até mesmo escalabilidade.

3.12. XCTest

O XCTest é um *framework* de teste desenvolvido pela Apple e nativo do Xcode. A estrutura XCTest serve para escrever testes de unidade para projetos que se integram com o fluxo de trabalho de teste do Xcode.

Os testes verificam se certas condições são satisfeitas durante a execução do código e registram as falhas de teste (com mensagens opcionais) se essas condições não forem satisfeitas. Os testes também podem medir o desempenho de blocos de código para verificar as regressões de desempenho e podem interagir com a interface do usuário de um aplicativo para validar os fluxos de interação do usuário (XCTEST, 2021).

3.13. EarlGrey

EarlGrey é uma estrutura/*framework* de teste de automação de interface do iOS desenvolvida pela Google que permite escrever testes claros e concisos.

A estrutura do EarlGrey, tem acesso a recursos de sincronização aprimorados. Ou seja, o EarlGrey sincroniza os testes automaticamente com a interface do usuário, além de executar solicitações de rede. Permite também implementar manualmente tempos personalizados, se necessário.

Os recursos de sincronização da EarlGrey ajudam a garantir que a interface do usuário esteja em um estado estável antes que as ações sejam realizadas. Isso aumenta muito a estabilidade do teste e torna os testes altamente reprodutíveis.

EarlGrey trabalha em conjunto com a estrutura XCTest e se integra com o Test Navigator do Xcode sendo possível executar testes diretamente do Xcode ou da linha de comando (EARLGREY, 2020).

3.14. The New Life

O The New Life é um aplicativo desenvolvido pela empresa Jungle Devs que tem como objetivo principal ajudar empresas a incentivar um estilo de vida mais saudável entre seus funcionários.

Neste aplicativo o usuário deve inicialmente criar uma conta cadastrando alguns dados pessoais no aplicativo. Após estar registrado no aplicativo, o usuário começa a receber diariamente alguns desafios que devem ser completados ao longo do dia. À medida em que o usuário vai completando esses desafios, ele vai pontuando.

Foi em cima deste aplicativo que os testes automatizados foram desenvolvidos.

Figura 2 - Logo do aplicativo The New Life

Fonte: Autoria própria

3.15. API

Uma API *(Aplication Programming Interface*) trata-se de um conjunto de rotinas, protocolos, pré-estabelecidos e documentados por uma determinada aplicação para que outras aplicações ou programas consigam utilizar as funcionalidades desta aplicação.

Desta forma entende-se que as APIs permitem uma interoperabilidade entre aplicações e ou comunicação entre aplicações e usuários (MULESOFT, 2021).

Esta interoperabilidade entre aplicações é realizada por chamadas (comandos), os mais comuns são:

- Get: Recebe informações para uma aplicação.
- Post: Envia informações para uma aplicação.
- Delete: Deleta informações.

4. METODOLOGIA

O presente trabalho foi dividido em etapas de desenvolvimento. A primeira etapa realizou-se uma pesquisa conceitual para melhor entendimento do problema e de como e quais maneiras poderiam ser solucionados. Em seguida realizou-se uma pesquisa informativa para identificar quais ferramentas de testes automatizados compatíveis com iOS existem disponíveis.

Na segunda etapa do desenvolvimento foram selecionadas duas ferramentas de teste e realizou-se um comparativo entre elas.

4.1. Pesquisa de ferramentas de teste disponíveis

Durante o processo de desenvolvimento do trabalho, realizaram-se pesquisas para descobrir quais ferramentas existem disponíveis no mercado, capazes de realizar testes de integração para a plataforma iOS.

A partir das pesquisas realizadas, descobriu-se que para a plataforma iOS existe um número limitado de ferramentas que atendem as necessidades do teste de integração, sendo que a maioria delas são voltadas aos testes de interface.

Por fim, foram selecionadas duas ferramentas de caráter *open source*, que possuem comunidade ativa atuando no desenvolvimento contínuo e na manutenção dessas ferramentas, de fácil implementação e que melhor atendem às necessidades a serem testadas. Após a seleção, as duas ferramentas escolhidas foram aplicadas no *software*, testando as mesmas funções e então realizou-se um comparativo entre as duas ferramentas para eleger a que melhor cumpre os critérios avaliados nos testes de integração.

As ferramentas avaliadas foram o EarlGrey e o XCTest. Essas ferramentas foram inicialmente pré-selecionadas por serem mais populares e bem ranqueadas pela comunidade na área de testes automatizados além de serem fáceis de implementar. Abaixo fez-se a análise comparativa entre as duas ferramentas.

4.2. Critérios analisados para escolha da ferramenta de teste

Os critérios analisados que foram determinísticos para a escolha da ferramenta de teste mais adequada são:

- Configuração inicial da ferramenta;
- Usabilidade;
- Sintaxe;
- Velocidade.

4.2.1. Análise comparativa da configuração inicial de cada ferramenta

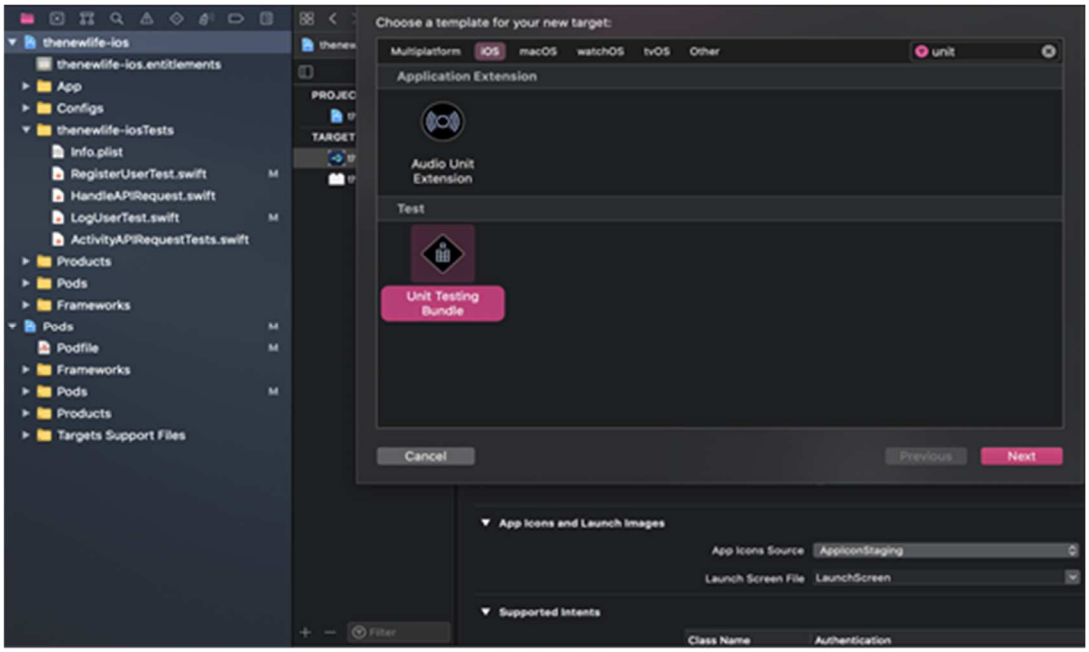
A adição do EarlGrey ao projeto pode ser feita de duas formas diferentes, a primeira seria adicionando o *framework* manualmente e a segunda usando o gerenciador de dependências CocoaPods. O método utilizado neste trabalho foi utilizando o gerenciador de dependências CocoaPods, por ser mais simples e rápido.

Instalação usando o CocoaPods como gerenciador de dependências:

O EarlGrey requer um alvo de teste. Como o EarlGrey adiciona alterações ao esquema e às fases de construção do alvo de teste, faz-se necessário criar um alvo de teste (figura 8) separado para adicionar testes EarlGrey. Adicionar um novo target ao projeto pode ser feito da seguinte forma:

1. Entrar nas configurações gerais do projeto no XCode;
2. Clicar no botão + que está na parte inferior da lista de destino;
3. Selecionar a opção iOS Unit Testing Bundle na seção Teste;
4. Fornecer as informações necessárias e clicar em Concluído.

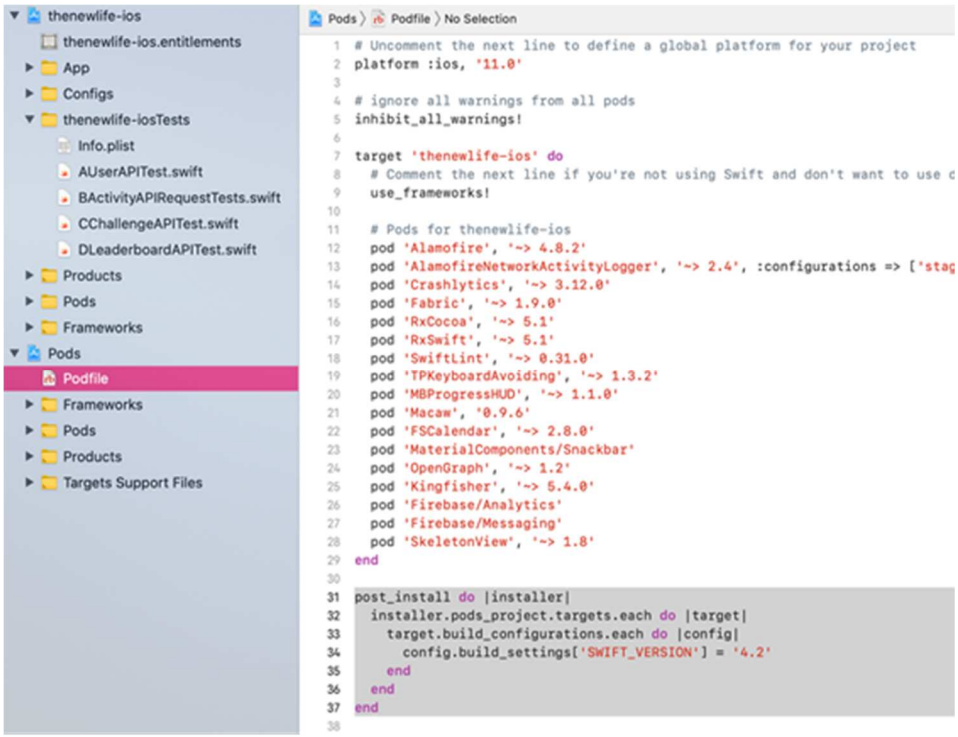
Figura 3 - Criação de um novo alvo de desenvolvimento



Fonte: Autoria própria

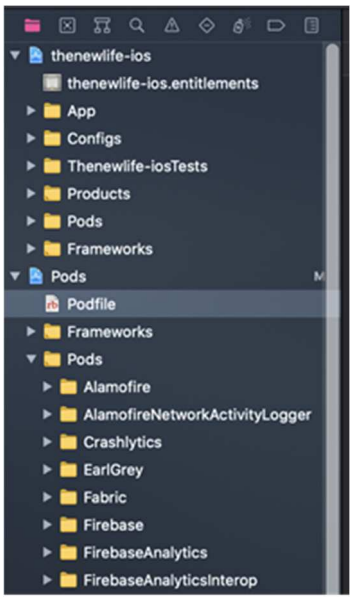
1. Após configurar um alvo de teste, precisa-se adicionar o EarlGrey como uma dependência de estrutura (Figura 4). Para fazer isso, faz-se necessário.
2. Adicionar o EarlGrey como uma dependência de teste ao Podfile do projeto (Figura 3).
3. Abrir o terminal do computador, navegar até a pasta do projeto e executar "gem install earlgrey".
4. Depois de ter instalado as gems corretamente, execute "pod install".
5. Para finalizar, verifique se o EarlGrey foi corretamente instalado na árvore do projeto (figura 4).

Figura 4 - Adição do EarlGrey no projeto



Fonte: Autoria própria

Figura 5 - Árvore do projeto



Fonte: Autoria própria

Para a instalação do XCTest basta apenas criar um novo ambiente de testes. É o mesmo procedimento descrito na primeira etapa da instalação do EarlGrey (figura 3).

1. Entrar nas configurações gerais do projeto no XCode;
2. Clicar no botão + que está na parte inferior da lista de destino;
3. Selecionar a opção iOS Unit Testing Bundle na seção Teste;
4. Fornecer as informações necessárias e clicar em Concluído.

Em seguida o Xcode irá gerar um alvo, alguns arquivos padronizados como Info.plist e um arquivo padrão para teste inicial.

4.2.1.1. Conclusão a partir dos métodos de instalação

Como o XCTest é uma ferramenta já integrada ao XCode, sua instalação se deu de forma mais simplificada do que o método requisitado pelo EarlGrey, onde faz-se necessária a importação de uma nova biblioteca ao projeto.

4.2.2. Análise comparativa entre a usabilidade das ferramentas

Para a realização deste comparativo, um teste de integração foi escrito utilizando a ferramenta XCTest e outro teste de integração equivalente utilizando a ferramenta EarlGrey.

O teste realizado para fazer o comparativo entre as ferramentas tem por objetivo verificar se ao fornecer dadas informações a função responsável por registrar um novo usuário ao aplicativo, a função deve retornar uma resposta válida. Os dois testes são demonstrados em seguida.

Primeiramente foram criadas algumas variáveis (Figura 6) simulando as informações inseridas pelo usuário para criação de uma conta no aplicativo. Em seguida descreve-se a função de teste responsável por criar um novo usuário no aplicativo. A figura 7 descreve a função teste usando o XCTeste e a figura 8 descreve a função teste usando o EarlGrey.

Figura 6 - Variáveis para cadastro de um novo usuário

```
let timeSeconds = TimeInterval(60)

let corporate = "Isis"
var username: String?
var email: String?
let password = "12345678"
let confirmPassword = "12345678"
```

Fonte: Autoria própria

Figura 7 - Código referente a primeira parte do registro de um novo usuário no aplicativo The New Life usando o XCTest

```
func test_A_RegisterNewUser() {

    username = "Test\(randomString(length: 6))"
    email = "\(randomString(length: 8))@test.com"

    let expectation = self.expectation(description: "Register new user")

    UserAPIManager
        .RegisterUser(email: email!,
                      name: username!,
                      password: password,
                      confirmPassword: confirmPassword,
                      corporate: corporate).request { [weak self] result in
                        guard self != nil else { return }
                        switch result {
                        case .failure(let error):
                            log(error)
                        case .success(let response):

                            SessionHelper.shared.createSession(with: response.user, token: response.token)
                            XCTAssertTrue(response.user.corporate == self?.corporate)
                            expectation.fulfill()
                        }
                      }
    wait(for: [expectation], timeout: timeSeconds)
}
```

Fonte: Autoria própria

Figura 8 - Código referente a primeira parte do registro de um novo usuário no aplicativo The New Life usando o EarlGrey

```
func test_A_RegisterUser() {

    var hasCompleted = false
    var output: UserAndToken?
    let myCondition = GREYCondition(name: "my condition") {
        return hasCompleted
    }
    UserAPIManager
        .RegisterUser(email: email,
                      name: username,
                      password: password,
                      confirmPassword: confirmPassword,
                      corporate: corporate).request { [weak self] result in
                        guard self != nil else { return }
                        switch result {
                        case .failure(let error):
                            log(error)
                            hasCompleted = true
                        case .success(let response):
                            log(response)
                            SessionHelper.shared.createSession(with: response.user, token: response.token)
                            output = response
                            hasCompleted = true
                        }
                      }
    _ = myCondition.waitWithTimeout(seconds: timeSeconds)
    GREYAssertTrue(output?.user.name == username, reason: "")
}
```

Fonte: Autoria própria

Diferentemente do XCTest o EarlGrey precisa ser alertado quando a chamada para API foi finalizada. Por conta disso a variável "hasCompleted" precisou ser criada. Além disso, o EarlGrey não consegue executar o teste dentro da função de chamada da API, apenas consegue executar o teste assim que a chamada foi finalizada. Então para que o teste fosse possível, a variável "output" foi criada.

A variável "output" é inicialmente declarada e atribui-se a ela um valor nulo, no caso de sucesso é atribuído o valor de resposta da chamada da API a esta variável e ao final da função faz-se um comparativo para verificar se o valor atribuído à variável equivale ao valor esperado.

Com o EarlGrey não se faz necessário esperar pelas "expectations" como no XCTest. Entretanto o EarlGrey espera que o usuário forneça uma condição de teste e que ela tenha um limite de tempo para que seja preenchida.

4.2.2.1. Conclusão a partir da usabilidade das ferramentas

A usabilidade do XCTest e do EarlGrey demonstraram ser similares, entretanto a usabilidade do XCTest aparentou ser mais simplificado pelo fato de não se fazer necessário criar variáveis para determinar o fim da chamada da API e também a possibilidade de executar o teste no sucesso da função.

4.2.3. Análise comparativa entre a velocidade das ferramentas

A velocidade para execução de todos os testes referentes ao registro e configuração de um novo usuário para o aplicativo, foi bastante similar usando as duas ferramentas.

O comparativo foi realizado de forma a executar cinco vezes os testes usando cada uma das ferramentas. Os resultados dos testes estão listados na tabela abaixo.

Tabela 1 - Tabela comparativa entre as velocidades de execução de testes das ferramentas EarlGrey e XCTest

Teste	EarlGrey	XCTest
1	4.272s	3.640s
2	3.374s	3.600s
3	4.219s	4.004s
4	5.004s	4.543s
5	4.326s	5.072s
Media dos tempos:	4.239s	4.172s

Fonte: Autoria própria

4.2.3.1. Conclusão a partir da velocidade das ferramentas

A ferramenta XCTest demonstrou ser 0.067 segundo mais rápida do que o EarlGrey. Demonstrando que a diferença de velocidade entre as duas ferramentas não é um fator tão significativo, pois as duas mostram serem similarmente rápidas.

4.2.4. Análise comparativa entre a sintaxe das ferramentas ao executar os mesmos testes

Ao importar o XCTest no projeto, várias funções de teste ficam disponíveis para execução dos testes. No presente projeto foram utilizadas as funções definidas como "Boolean Assertions" (figura 9) onde pode-se verificar se a comparação entre as respostas fornecidas pelo servidor ao executar os testes, quando comparadas as informações dadas pelo usuário, são verdadeiras ou falsas.

Figura 9 - Exemplo de sintaxe do XCTest

```
XCTAssertTrue(response.user.corporate == self?.corporate)
```

Fonte: Autoria própria

O EarlGrey ao ser importado ao projeto também fornece uma biblioteca própria com várias funções que podem ser usadas na validação dos testes. Assim como no XCTest, as funções utilizadas nos testes do EarlGrey foram do tipo "Boolean Assertions" (figura 10).

Figura 10 - Exemplo de sintaxe EarlGrey

```
GREYAssertTrue(output?.user.name == username, reason: "")
```

Fonte: Autoria própria

4.2.4.1. Conclusão a partir da sintaxe das ferramentas

As duas ferramentas demonstraram ser bem similares quanto à sintaxe. Cada uma delas apresenta uma biblioteca própria contendo funções específicas capazes de realizar os testes de integração e fazer um comparativo com o resultado esperado.

4.3 Cenários de teste

Os testes foram realizados nas dependências da empresa Jungle Devs onde foram utilizadas as seguintes configurações:

Hardware: Servidor virtual AWS, MacBook Pro, ativos de rede (roteadores, *modens* e *switch*).

Software: Sistema operacional Mac OS, XCode, API.

4.3.1 Teste de integração

Com o cenário configurado foram realizados comandos no ambiento do XCode com a finalidade de verificar se os dados contidos no servidor correspondiam com os dados esperados. Neste foi verificado a integralidade entre o banco de dados e as informações fornecidas pelo usuário do aplicativo, da seguinte forma, foi criada uma tabela no banco de dados contendo informações do usuário, a exemplo, nome, nome da empresa em que trabalha, data de nascimento, altura, peso e gênero. Fez-se uma chamada para a API do tipo *post* para registrar os dados fornecidos pelo usuário por meio da interface do usuário. Fez-se também chamadas para a API do tipo *get* para popular o aplicativo com os requisitos da solução e também com as informações inseridas pelo usuário.

Com o intuito de verificar a integralidade dos registros fez-se uma validação por meio de funções de asserção. Asserções são funções inseridas no código do programa que servem para verificar se determinadas condições são verdadeiras nas funcionalidades do programa.

5. SOLUÇÃO

Sempre que os testes são programados para acontecer um usuário novo é criado automaticamente. Isso foi possível pois foi criada uma função que cria um e-mail e um nome aleatoriamente. Além disso, fez-se uso do "*SessionHelper*" que armazena informações quando o usuário é criado. Com o "*SessionHelper*" é possível passar a mesma informação para outras classes do aplicativo e por consequência testar outras funções no código sem que as variáveis armazenadas sejam reiniciadas.

O XCode executa os testes por ordem alfabética, por isso uma abordagem utilizada neste trabalho foi colocar uma letra em destaque no nome das funções pois assim consegue-se informar ao programa qual a primeira função que deve ser executada.

É de fundamental importância que a primeira função a ser executada seja a função responsável por registrar um novo usuário, pois é nessa função que o programa obtém as credenciais necessárias para executar as outras funções.

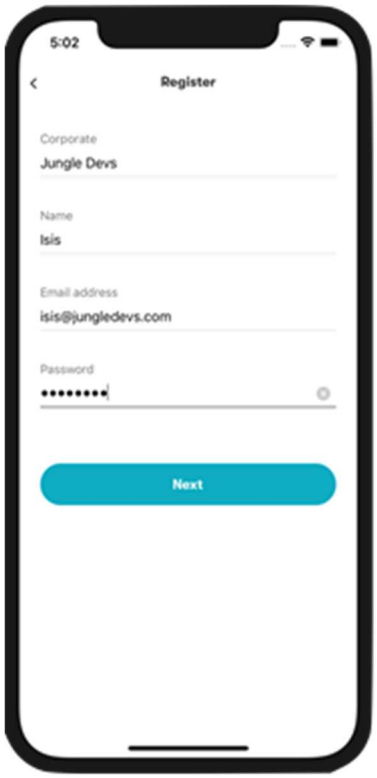
Para realização dos testes de integração, foram escritas todas as funções que requisitam uma resposta de um servidor externo. As funções de teste foram separadas em quatro documentos com intuito de deixar o código mais organizado.

No primeiro documento foram escritos os testes relacionados ao cadastro e login de usuários. O segundo documento está associado ao registro de atividades do usuário. O terceiro trata dos desafios que o usuário recebe ao longo dos dias e como interage com eles e o quarto e último documento está relacionado com os pontos adquiridos pelo usuário e sua colocação no aplicativo.

5.1 Documento 1

O primeiro documento é referente ao registro de um novo usuário no aplicativo e também ao login do usuário. O primeiro documento pode ser visualizado na íntegra no apêndice A

Figura 11 - Registro dos usuários do aplicativo The New Life - parte 1



Fonte: Autoria própria

Para que o registro de um novo usuário seja válido, o usuário deve fornecer algumas informações. A figura 11 ilustra o formato de objeto que deve ser válido ao se tentar registrar um novo usuário.

A função descrita na figura 12 cria um nome aleatório e um e-mail aleatório para registrar um novo usuário cada vez que os testes rodarem. Nessa primeira parte o usuário deve fornecer um nome, e-mail, nome da empresa em que trabalha e uma senha. Se o nome da empresa que o usuário forneceu estiver registrada no banco de dados do aplicativo e os campos de e-mail e senha forem validados, o usuário poderá passar para a segunda etapa do registro da conta.

Além de criar um usuário, a função irá verificar se o objeto de resposta fornecido pela função é o mesmo que a informação fornecida pelo usuário inicialmente.

Essa verificação é feita pelo comando ilustrado na figura 12. Se a verificação for verdadeira o teste é válido, se as respostas forem diferentes o teste é reprovado.

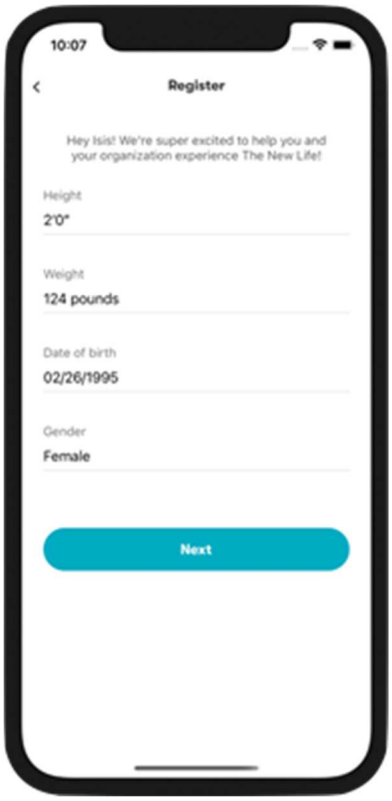
A figura 12 ilustra a segunda função de teste responsável por fazer a segunda parte do registro do usuário, onde o usuário fornece ao aplicativo informações mais detalhadas sobre seu perfil, são elas: altura, peso, data de nascimento e gênero.

Figura 12 - Código referente a segunda parte do registro do usuário

```
func test_B_UserRegisterDetails() {

    let expectation = self.expectation(description: "Register user details")

    username = SessionHelper.shared.currentUser?.name
    email = SessionHelper.shared.currentUser?.email

    UserAPIManager
        .UserDetails(height: height,
                     weight: weight,
                     dateOfBirth: dateOfBirth,
                     gender: gender)
        .request { [weak self] result in
            guard let self = self else { return }
            print(result)
            switch result {
            case .failure(let error):
                log(error)
            case .success(let response):
                XCTAssertTrue(response.name == self.username)
                XCTAssertTrue(response.email == self.email)
                expectation.fulfill()
            }
        }
    wait(for: [expectation], timeout: timeSeconds)
}
```

Fonte: Autoria própria

Figura 13 - Registro dos usuários do aplicativo The New Life - parte 2



Fonte: Autoria própria

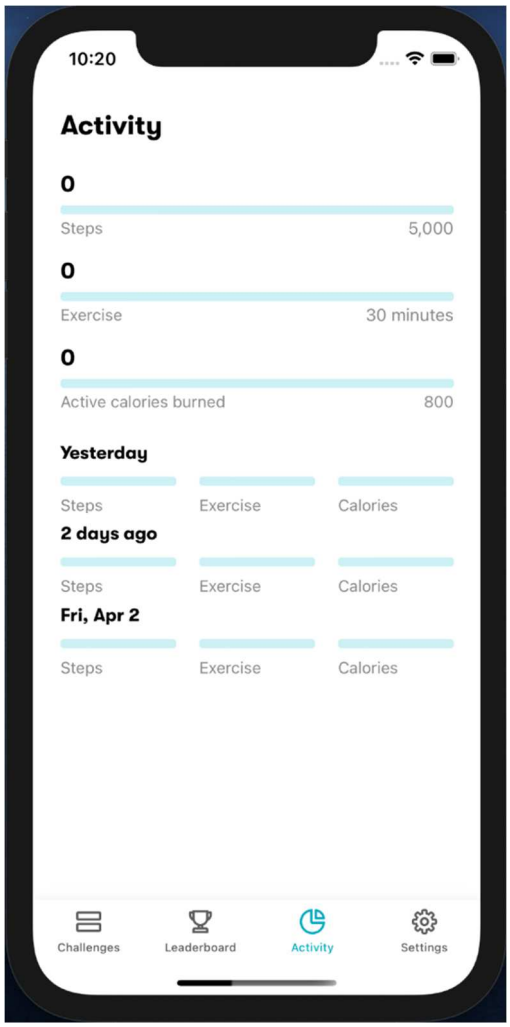
Assim que o usuário fornece as informações necessárias na segunda etapa do registro, e apertar o botão "Next", o aplicativo fará uma nova chamada para a API de forma a pedir para que o servidor associe essas novas informações aos dados registrados na primeira etapa do processo de registro de um novo usuário.

Na função de registro de detalhes do usuário, também é verificado se a resposta fornecida pelo servidor é equivalente às informações fornecidas pelo usuário anteriormente. Se os dados forem equivalentes, o teste é validado, se não o teste é reprovado.

5.2 Documento 2

O documento 2 é referente ao registro de atividades usuário (figura 14) registradas no aplicativo. As atividades estão sincronizadas com o Apple Health que é o aplicativo nativo do iPhone que armazena dados de saúde do usuário. O documento 2 pode ser conferido na íntegra no apêndice B.

Figura 14 – Interface de atividades do usuário



Fonte: Autoria própria

A função de teste "test_A_fetchFromHealthKit" (figura 15) verifica por intermédio do comando "XCTAssertTrue" se a resposta fornecida pelo servidor é equivalente aos dados fornecidos à função. Se a comparação for verdadeira, o teste é validado.

Figura 15 - Função teste referente às atividades do usuário

```
func test_A_fetchFromHealthKit() {
    let expectation = self.expectation(description: "Fetch Activity From Health Kit")

    ActivityAPIManager.CreateActivity.init(steps: steps, exercise: exercise, calories: calories, date: Date())
        .request { result in
            switch result {
            case .failure(let error):
                log(error.localizedDescription)
            case .success(let response):
                XCTAssertTrue(response.calories == self.calories)
                XCTAssertTrue(response.exercise == self.exercise)
                XCTAssertTrue(response.steps == self.steps)

                expectation.fulfill()
            }
        }

    wait(for: [expectation], timeout: timeSeconds)
}
```

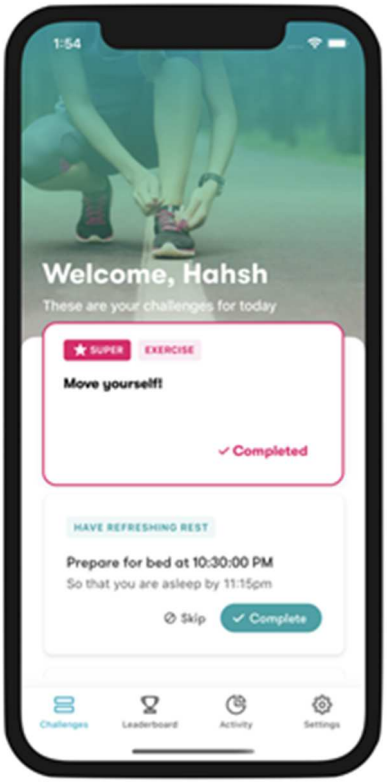
Fonte: Autoria própria

5.3 Documento 3

O documento 3 refere-se aos desafios propostos aos usuários. O aplicativo tem como funcionalidade propor diariamente 10 desafios a seus usuários que devem ser completados ao longo do dia. Se o desafio for completado, o usuário deve apertar o botão "Complete", caso o usuário opte por não fazer o desafio, o botão "*Skip*" deve ser pressionado.

O documento 3 pode ser conferido na íntegra no apêndice C.

Figura 16 – Interface de desafios do usuário



Fonte: Autoria própria

A imagem x ilustra a função que requisita os desafios diários do usuário para o servidor. Essa função deve retornar um array contendo dez desafios/challenges.

Para validação do teste, foi realizada uma comparação com a finalidade de verificar se a resposta do servidor contém dez itens, se a verificação for verdadeira, o teste será validado.

Figura 17 - Função teste que requisita os desafios do usuário

```
func test_H_FetchTodayChallenges() {
    let expectation = self.expectation(description: "Request for user")
    ChallengesAPIManager
        .FetchChallenges()
        .request { result in
            switch result {
            case .failure(let error):
                log(error.localizedDescription)
            case .success(let todayChallenges):
                SessionHelper.shared.createChallengesSession(with: todayChallenges.challenges)
                XCTAssertTrue(todayChallenges.challenges.count == 10)
                expectation.fulfill()
            }
        }
    wait(for: [expectation], timeout: timeSeconds)
}
```

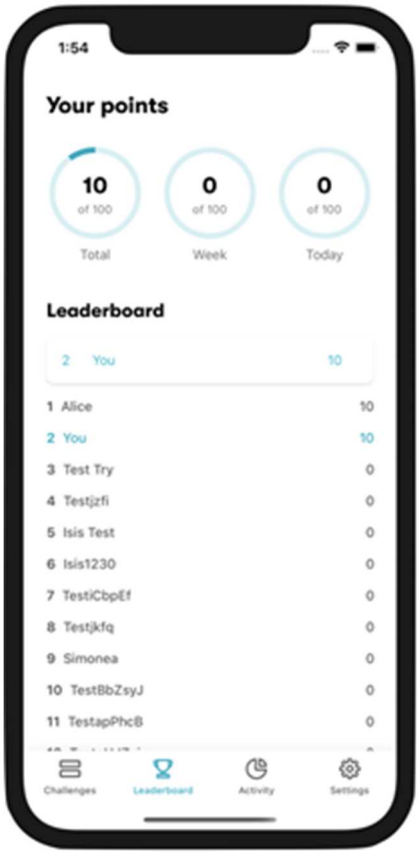
Fonte: Autoria própria

5.4 Documento 4

O documento 4 refere-se a pontuação e ranking dos usuários do aplicativo existentes em uma mesma empresa. Sempre que os testes são executados, um novo usuário é criado, cada usuário tem inicialmente zero pontos.

O documento 4 pode ser conferido na íntegra no apêndice D.

Figura 18 – Interface de pontuação do usuário



Fonte: Autoria própria

A figura 19 ilustra a função que requisita a pontuação do usuário e o placar das pessoas cadastradas em uma mesma empresa.

Sabendo que cada usuário criado inicializa com zero pontos, foi realizada uma comparação com a finalidade de verificar se a resposta do servidor, quando requisita a pontuação do usuário, equivale a zero, se a verificação for verdadeira, o teste será validado.

Figura 19 - Função de teste que requisita a pontuação do usuário

```
func test_fetchUserScore() {
    let expectation = self.expectation(description: "Request for user")
       LeaderboardAPIManager
           .FetchUserScore()
           .request { result in
               switch result {
               case .failure:
                   print(result)

               case .success(let response):
                XCTAssertTrue(response.userScore.total == 0)
                expectation.fulfill()
               }
       }
    wait(for: [expectation], timeout: timeSeconds)
   }
```

Fonte: Autoria própria

5.5. Comparativo entre testes manuais e testes com XCTEST

Após a execução de todos os testes, realizou-se um teste comparativo entre os testes manuais e os testes utilizando a ferramenta XCTest.

Utilizando o XCTest foi possível executar quinze testes em 6.453 segundos. Os resultados dos testes (figura 20) estão demonstrados na imagem abaixo.

Figura 20 - Resultado dos testes

```
Test Case '-[thenewlife_iosTests.DLeaderboardAPITest test_fetchUserScore]' passed (0.300 seconds).
Test Suite 'DLeaderboardAPITest' passed at 2020-10-24 20:44:03.540.
     Executed 1 test, with 0 failures (0 unexpected) in 0.300 (0.300) seconds
Test Suite 'thenewlife-iosTests.xctest' passed at 2020-10-24 20:44:03.541.
     Executed 15 tests, with 0 failures (0 unexpected) in 6.436 (6.453) seconds
Test Suite 'Selected tests' passed at 2020-10-24 20:44:03.541.
     Executed 15 tests, with 0 failures (0 unexpected) in 6.436 (6.454) seconds
```

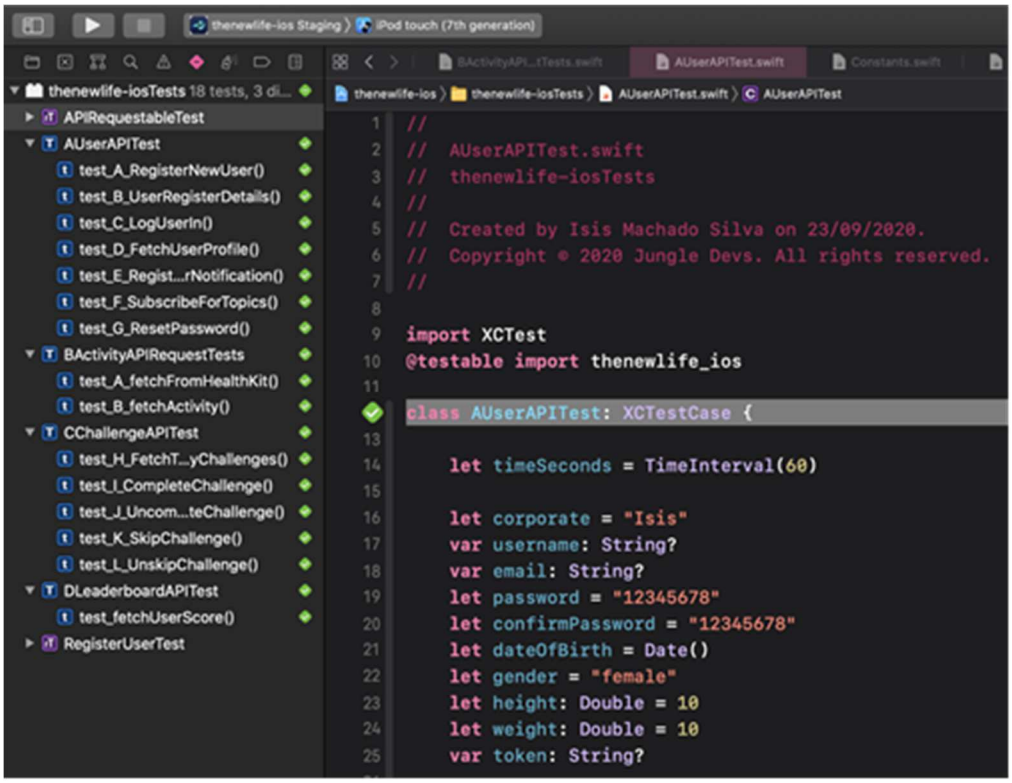
Fonte: Autoria própria

Os testes manuais foram executados de forma que o desenvolvedor interage com o aplicativo, passando por todas as funções que requerem integração com a API e essa interação foi cronometrada. O teste levou um total de 1:52,40 min para ser completamente executado.

O Xcode providencia uma interface (figura 21) amigável para controle dos testes, fazendo-se possível monitorar cada função de teste executada. Na imagem abaixo é possível verificar todos os testes escritos e devidamente organizados por ordem alfabética.

O diamante verde é um indicativo de que o teste teve sucesso.

Figura 21 – Interface de teste do Xcode



Fonte: Autoria própria

6. CONCLUSÃO

O presente trabalho avaliou a importância da utilização de testes automatizados no desenvolvimento de aplicativos iOS, tendo como objetivo otimizar o processo de desenvolvimento de software na empresa Jungle Devs.
Com este trabalho, pôde-se verificar que a utilização de testes automatizados no desenvolvimento de aplicativos, faz com que os testes sejam executados mais rapidamente, possibilitando a verificação de erros integrados ao projeto com maior facilidade e rapidez.

Pôde-se verificar também que a ferramenta XCTest, nativa do XCode e desenvolvida pela Apple, mostrou ser uma ótima ferramenta para execução de testes de integração, além de ser de fácil configuração, de uso e também, rápida.

Com as funções desenvolvidas neste trabalho, foi possível executar todos os testes de integração desenvolvidos no projeto a partir de um único comando, contribuindo para a otimização dos testes.

Fazendo uma análise comparativa com os testes realizados manualmente e automaticamente, pôde-se verificar que os testes manuais levaram em média 2 minutos para testar todas as funções de integração existentes no aplicativo, enquanto os testes automatizados somatizam 6.45 segundos para execução de todos os testes. Pode-se concluir então que os testes automáticos são 18 vezes mais rápidos que os testes manuais.

O presente trabalho demonstrou ser de grande importância para a empresa Jungle Devs, pois possibilita desenvolver produtos com maior qualidade e confiabilidade de forma mais rápida e otimizada. Além de poupar o desenvolvedor de ter que desperdiçar tempo em retrabalho do projeto, pois as falhas de integração são rapidamente encontradas a partir do momento em que os testes são executados.

Verificar falhas mais rapidamente implica também em uma economia para a empresa, visto que os projetos são monetizados com base nas horas trabalhadas, ou seja, o cliente paga por uma determinada aplicação que deve ser executada em um determinado período de tempo, caso o desenvolvedor demore mais para entregar a aplicação, mais caro o projeto vai se tornando.

Pretende-se, no futuro, aplicar testes automatizados em outros produtos da empresa.

REFERÊNCIAS BIBLIOGRÁFICAS

ALVES, Arthur Moreira. **Comparativo de ferramentas de automatização de testes: Selenium IDE e Selenium WebDriver**. Orientador: José Eduardo Ferreira Lopes. 2018. 29 f. TCC (Graduação) – Bacharelado em Gestão da Informação, Universidade Federal de Uberlândia, 2018. Disponível em: <https://repositorio.ufu.br/bitstream/123456789/28000/1/ComparativoFerramentasAutomatização.pdf>. Acesso em: 1 mar. 2021.

BERTOLI, Sandra Maria de Souza. **Guia de Segurança para Dispositivos Móveis: Hardware, Software e Comportamento**. Orientador: Carlos A. Maziero. 2014. 77 f. TCC (Graduação) – Curso de Tecnologia em Sistemas para Internet, Departamento Acadêmico de Informática, Universidade Tecnológica Federal do Paraná, 2014. Disponível em: http://repositorio.utfpr.edu.br/jspui/handle/1/9810. Acesso em: 1 mar. 2021.

COCOAPODS. **What Is Cocopods**. [S.l] [2021?]. Disponível em: <https://cocoapods.org> Acesso em: 10 fev. 2021.

CRISPIN, L. **Agile Testing: A Practical Guide for Testers and Agile Teams**. [S.l.]: Addison-Wesley Professional, 2009. Acesso em: 1 mar. 2021.

CUNHA, Simone Moreira. **Engenharia de software - Uma abordagem às fases de teste**. Orientador: Ângelo Moura Guimarães. 2010. 106 f. Monografia – Curso de Especialização em Informática, Departamento de Ciências Exatas, Universidade Federal de Minas Gerais, 2010. Disponível em: https://repositorio.ufmg.br/bitstream/1843/BUBD-B8XKUP/1/espinform_tica_simonemoreiracunha_monografia.pdf. Acesso em: 1 mar. 2021.

EARLGREY. **EarlGrey**. [S.l] [2020?]. Disponível em: <https://github.com/google/EarlGrey> Acesso em: 10 fev. 2021.

OPENSOURCE. **The Open Source Definition**. [S.l] [2007]. Disponível em:

< https://opensource.org/docs/osd > Acesso em: 10 jan. 2021.

PRESSMAN, R. S. Engenharia de Software. 6. ed. São Paulo: McGraw-Hill, 2006.

GOODENOUGH, J.B · 1975 · ACM Symp. on Principles of Programming Languages, Palo, Alto, Calif., Jan. *1975*, pp

LINDLEY, Cody. **Front-End Developer Hand Book**. 2018. Disponível em: <https://frontendmasters.com/books/front-end-handbook/2018/ >. Acesso em: 04 mai. 2021.

MULESOFT. **What is na API?**. Disponível em: <https://www.mulesoft.com/resources/api/what-is-an-api> Acesso em: 1 mai. 2021.

ROSENCRANE, Linda. **Software**. TechTarget, mar 2021. Disponível em: <https://searchapparchitecture.techtarget.com/definition/software> Acesso em: 1 mar. 2021.

SCHWABER, J. S. K. **The Definitive Guide to Scrum: The Rules of the Game. The Scrum Guide**. [S.l] [2020] Disponível em: <https://scrumguides.org/scrum-guide.html> Acesso em: 13 mar. 2021.

SWIFT. **About Swift**. [S.l] [2021]. Disponível em: <https://swift.org/about/#platform-support> Acesso em: 10 fev. 2021.

XCODE. **XCode**. [S.l] [2021]. Disponível em: <https://developer.apple. com/xcode/> Acesso em: 10 fev. 2021.

XCTEST. **XCTest**. Disponível em: <https://developer.apple.com/documentation/xctest> Acesso em: 10 fev. 2021.

YAMAMOTO, Ígor Assis Rocha. **Desenvolvimento de Aplicativo para Locação de Academias de Musculação Utilizando Testes Automatizados de Interface Gráfica**. Orientador: Rômulo Silva de Oliveira. 2018. 90 f. PFC (Graduação) – Curso

de Engenheira de Controle e Automação, Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis, 2018. Disponível em: <https://repositorio.ufsc.br/handle/123456789/200012>. Acesso em: 1 mar. 2021.

APÊNDICE

APÊNDICE A — Documento referente a criação de um novo usuário no aplicativo The New Life

```
//  AUserAPITest.swift
//  thenewlife-iosTests
//  Created by Isis Machado Silva on 23/09/2020.
//  Copyright © 2020 Jungle Devs. All rights reserved.

import XCTest
@testable import thenewlife_ios

class AUserAPITest: XCTestCase {
   let timeSeconds = TimeInterval(60)
   let corporate = "Isis"
   var username: String?
   var email: String?
   let password = "12345678"
   let confirmPassword = "12345678"
   let dateOfBirth = Date()
   let gender = "female"
   let height: Double = 10
   let weight: Double = 10
   var token: String?

   private var subscribedTopics: [String] = []
         private let notificationTopics = ["trust_more_and_stress_less", "have_refreshing_rest",
"enjoy_sunlight", "nutrition", "water", "live_temperately", "invest_in_others", "fresh_air", "education",
"super_challenge"]

   func randomString(length: Int) -> String {
     let letters = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
     return String((0..<length).map{ _ in letters.randomElement()! })
         }

         // MARK: - Tests

   func test_A_RegisterNewUser() {
      username = "Test\(randomString(length: 6))"
      email = "\(randomString(length: 8))@test.com"
      let expectation = self.expectation(description: "Register new user")
      UserAPIManager
         .RegisterUser(email: email!,
                  name: username!,
                  password: password,
                confirmPassword: confirmPassword,
                  corporate: corporate).request { [weak self] result in
                guard self != nil else { return }
                switch result {
                case .failure(let error):
                   log(error)
                case .success(let response):
                   SessionHelper.shared.createSession(with: response.user, token: response.token)
                   XCTAssertTrue(response.user.corporate == self?.corporate)
                   expectation.fulfill()
```

```
                }
                }
    wait(for: [expectation], timeout: timeSeconds)
        }

func test_B_UserRegisterDetails() {
    let expectation = self.expectation(description: "Register user details")
        username = SessionHelper.shared.currentUser?.name
        email = SessionHelper.shared.currentUser?.email
    UserAPIManager
        .UserDetails(height: height,
                     weight: weight,
                 dateOfBirth: dateOfBirth,
                     gender: gender)
        .request { [weak self] result in
         guard let self = self else { return }
         print(result)
         switch result {
         case .failure(let error):
            log(error)
         case .success(let response):
            XCTAssertTrue(response.name == self.username)
            XCTAssertTrue(response.email == self.email)
            expectation.fulfill()
        }
        }
    wait(for: [expectation], timeout: timeSeconds)
        }

func test_C_LogUserIn() {
    let expectation = self.expectation(description: "Request for user")
    let userEmail = (SessionHelper.shared.currentUser?.email)!
    UserAPIManager.Login(email: userEmail, password: password)
        .request { result in
         switch result {
         case .failure(let error):
            log(error)
         case .success(let response):
            SessionHelper.shared.createSession(with: response.user, token: response.token)
            XCTAssertTrue(response.user.email == userEmail)
            expectation.fulfill()
        }
        }
    wait(for: [expectation], timeout: timeSeconds)
        }

func test_D_FetchUserProfile() {
    let expectation = self.expectation(description: "Request for user")
    let userEmail = (SessionHelper.shared.currentUser?.email)!
    UserAPIManager.FetchUserProfile()
        .request { [weak self] result in
         guard self != nil else { return }
         switch result {
         case .failure(let error):
            log(error)
         case .success(let response):
            XCTAssertTrue(response.email == userEmail)
            expectation.fulfill()
```

```
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_E_RegisterForNotification() {
    let expectation = self.expectation(description: "Request for user")

UserAPIManager.AddDeviceToken.init(deviceToken: self.token)
            .request { [weak self] result in
              switch result {
              case .failure(let error):
                 log("Could not register device to receive push notifications: \(error)")
              case .success(let response):
                 XCTAssertNotNil(response)
                 expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_F_SubscribeForTopics() {
        let expectation = self.expectation(description: "Request for user")
        UserAPIManager.SubscribeTopics(with: [])
            .request { [weak self] result in
              guard let self = self else { return }
              switch result {
              case .failure(let error):
                 log(error)
              case .success(let response):
                 XCTAssertTrue(response.topics == [])
                 expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_G_ResetPassword() {
        let expectation = self.expectation(description: "Request for user")
        let userEmail = (SessionHelper.shared.currentUser?.email)!
        let userPassword = "12345678"
        UserAPIManager.ResetPassword.init(email: userEmail)
            .request { [weak self] result in
              guard let self = self else { return }
              switch result {
              case .failure(let error):
                 log(error)
              case .success(let response):
                 XCTAssertNotNil(response)
                 expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
    }
}
```

APÊNDICE B — Documento referente ao registro de atividades do usuário no aplicativo The New Life

```swift
//  ActivityAPIRequestTests.swift
//  thenewlife-iosTests
//  Created by Isis Machado Silva on 04/10/2020.
//  Copyright © 2020 Jungle Devs. All rights reserved.

import XCTest
@testable import thenewlife_ios

class BActivityAPIRequestTests: XCTestCase {
   var steps: Int = 10
   var exercise: Int = 100
   var calories: Int = 1000
   let date = Date()
   let timeSeconds = TimeInterval(5)
   func test_A_fetchFromHealthKit() {
      let expectation = self.expectation(description: "Fetch Activity From Health Kit")
      ActivityAPIManager.CreateActivity.init(steps: steps, exercise: exercise, calories: calories, date: Date())
           .request { result in
            switch result {
            case .failure(let error):
               log(error.localizedDescription)
            case .success(let response):
               XCTAssertTrue(response.calories == self.calories)
               XCTAssertTrue(response.exercise == self.exercise)
               XCTAssertTrue(response.steps == self.steps)
               expectation.fulfill()
           }
           }
      wait(for: [expectation], timeout: timeSeconds)
           }

   func test_B_fetchActivity() {
      let expectation = self.expectation(description: "Fetch Activity")
      ActivityAPIManager.FetchActivities()
           .request { activityListResponse in
            switch activityListResponse {
            case .failure(let error):
               log (error.localizedDescription)
            case .success(let response):
               XCTAssertTrue(response.results.first?.calories == self.calories)
               expectation.fulfill()
           }
           }
      wait(for: [expectation], timeout: timeSeconds)
   }
}
```

APÊNDICE C — Documento referente aos desafios do usuário do aplicativo The New Life

```
//  CChallengeAPITest.swift
//  thenewlife-iosTests
//  Created by Isis Machado Silva on 09/10/2020.
//  Copyright © 2020 Jungle Devs. All rights reserved.

import XCTest
@testable import thenewlife_ios

class CChallengeAPITest: XCTestCase {
    var id: Int?
    let timeSeconds = TimeInterval(60)
    let challenge = SessionHelper.shared.todayChallenges
    let user = SessionHelper.shared.currentUser
    func test_H_FetchTodayChallenges() {
        let expectation = self.expectation(description: "Request for user")
        ChallengesAPIManager
            .FetchChallenges()
            .request { result in
             switch result {
             case .failure(let error):
                log(error.localizedDescription)
             case .success(let todayChallenges):
                SessionHelper.shared.createChallengesSession(with: todayChallenges.challenges)
                XCTAssertTrue(todayChallenges.challenges.count == 10)
                expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_I_CompleteChallenge() {
        let expectation = self.expectation(description: "Request for user")
        ChallengesAPIManager
            .CompleteChallenge(id: challenge?.first?.id ?? 4256)
            .request { result in
             switch result {
             case .failure(let error):
                log(error.localizedDescription)
             case .success(let completedChallenge):
                XCTAssertTrue(completedChallenge.status == .completed)
                expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_J_UncompleteChallenge() {
        let expectation = self.expectation(description: "Request for user")
        ChallengesAPIManager
            .UncompleteChallenge(id: challenge?.first?.id ?? 4256)
            .request { result in
             switch result {
             case .failure(let error):
                log(error)
             case .success(let undoneChallenge):
```

```
                XCTAssertTrue(undoneChallenge.status == .pending)
                expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_K_SkipChallenge() {
        let expectation = self.expectation(description: "Request for user")
        ChallengesAPIManager
            .SkipChallenge(id: challenge?.first?.id ?? 4256)
            .request { result in
             switch result {
             case .failure(let error):
                log(error)
             case .success(let skippedChallenge):
                XCTAssertTrue(skippedChallenge.status == .skipped)
                expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
            }

    func test_L_UnskipChallenge() {
        let expectation = self.expectation(description: "Request for user")
        ChallengesAPIManager
            .UnskipChallenge(id: challenge?.first?.id ?? 4256)
            .request { result in
             switch result {
             case .failure(let error):
                log(error)
             case .success(let unskippedChallenge):
                log(unskippedChallenge)
                XCTAssertTrue(unskippedChallenge.status == .pending)
                expectation.fulfill()
            }
            }
        wait(for: [expectation], timeout: timeSeconds)
    }
}
```

APÊNDICE D — Documento referente a pontuação dos usuários do aplicativo The New Life

```
//  DLeaderboardAPITest.swift
//  thenewlife-iosTests
//  Created by Isis Machado Silva on 09/10/2020.
//  Copyright © 2020 Jungle Devs. All rights reserved.

import XCTest
@testable import thenewlife_ios

class DLeaderboardAPITest: XCTestCase {
    let timeSeconds = TimeInterval(60)
    func test_fetchUserScore() {
        let expectation = self.expectation(description: "Request for user")
          LeaderboardAPIManager
              .FetchUserScore()
              .request { result in
                  switch result {
                  case .failure:
                      print(result)
                  case .success(let response):
                   XCTAssertTrue(response.userScore.total == 0)
                   expectation.fulfill()
                          }
              }
        wait(for: [expectation], timeout: timeSeconds)
              }
}
```