UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE E INOVAÇÃO

LEVY MARCELO GOULART CUNHA

Automação de testes em aplicativos móveis

Monografia apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software e Inovação.

Orientador: Profª. Dra. Érika Cota

Porto Alegre

2021

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Cunha, Levy Marcelo Goulart
Automação de testes em aplicativos móveis / Levy Marcelo Goulart Cunha. – 2021.
62 f.: il.
Orientadora: Érika Fernandes Cota.
Monografia (Especialização) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2021.
1. Garantia da qualidade. 2. Automação de testes. 3. Interface do usuário. 4. Aplicativos nativos para dispositivos móveis. I. Fernandes Cota, Érika. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitor: Prof[a]. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof. Júlio Otávio Jardim Barcellos
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do Curso: Prof[a]. Karin Becker
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Agradeço a Deus por ter me proporcionado saúde para chegar aqui. Aos meus pais David e Eliane por me apoiarem ao longo da vida. A minha namorada Michele Wichrowski pela compreensão, ajuda e por ser essencial em todos os momentos. A minha orientadora Dra. Érika Cota pela dedicação e enorme ajuda durante o trabalho. Ao meu ex-colega Israel Ermel por me apoiar nas dúvidas em relação ao desenvolvimento Android. A todos que de alguma forma puderam contribuir para realização deste trabalho.

RESUMO

Na atual conjuntura os aplicativos para dispositivos móveis (*Smartphone*, *Tablets*, entre outros) tornaram-se indispensáveis no cotidiano, bem como diferencial no relacionamento do cliente com empresa. Garantir a qualidade sem perder agilidade durante o desenvolvimento de aplicações móveis é primordial, portanto, planejar e executar testes de *software* com a maior abrangência possível passa a ser indispensável. O uso da automação de testes demonstra seu valor, pois agrega rapidez diminuindo a quantidade de repetições necessárias, minimizando possíveis problemas, porém existem grandes desafios para este tipo de aplicação como variedade de dispositivos e versões, assim como prazos de entrega. Neste contexto o trabalho tem como aspecto principal uma análise comparativa de ferramentas para automação do teste funcional no nível de sistema (baseado na interface do usuário) em aplicativos nativos (sistema operacional Android e/ou iOS). Existem duas abordagens possíveis que se diferem pela adequação ao sistema operacional, então em um primeiro momento foi realizado um levantamento de diversas ferramentas e após uma avaliação de alguns parâmetros, foram selecionadas três ferramentas. A funcionalidade de cadastro em um aplicativo de uma empresa local foi utilizada, como estudo de caso para guiar a análise comparativa das ferramentas selecionadas. Como resultado, foram identificadas as vantagens e limitações de cada ferramenta no contexto daquela organização.

Palavras-chave: Garantia da qualidade. Automação de testes. Interface do usuário. Aplicativos nativos para dispositivos móveis.

Test automation in mobile apps

ABSTRACT

In the current situation, applications for mobile devices (Smartphone, Tablets, among others) have become indispensable in everyday life, as well as a differential in the relationship between the customer and the company. Ensuring quality without losing agility during the development of mobile applications is essential; therefore, planning and executing software tests with the greatest possible scope becomes essential. The use of test automation demonstrates its value, as it adds speed by reducing the amount of repetitions required, minimizing possible problems, but there are great challenges for this type of application as a variety of devices and versions, as well as delivery times. In this context, the work has as its main aspect a comparative analysis of tools for functional test automation at system level (based on the user interface) in native applications (Android and/or iOS operating system). There are two possible approaches that differ in their suitability to the operating system, so at first a survey of several tools was carried out and after an evaluation of some parameters, three tools were selected. The registration functionality in a local company application was used as a case study to guide the comparative analysis of the selected tools. As a result, the advantages and limitations of each tool in the context of that organization were identified.

Keywords: Quality Assurance. Test automation. User Interface. Native apps for mobile devices.

LISTA DE FIGURAS

Figura 1.1 – Diagrama de exemplo das abordagens de automação de testes em aplicativos nativos..10
Figura 2.1 – Os três tipos de aplicativos para dispositivos móveis..............................13
Figura 2.2 – Pirâmide de teste..17
Figura 2.3 – Exemplos de ferramentas na pirâmide de teste......................................18
Figura 3.1 – Tela inicial do Unicred Mobile...21
Figura 3.2 – Primeira tela do Onboarding Digital (tela de cadastro).............................22
Figura 3.3 – Tela: O que é um nome social?..23
Figura 4.1 – Arquitetura do Appium...30
Figura 5.1 – Pacote de teste de interface do usuário no XCTest................................36
Figura 5.2 – Diagrama da arquitetura de teste...37
Figura 5.3 – Primeira tela do aplicativo desenvolvido...40
Figura 5.4 – Segunda tela do aplicativo desenvolvido..41

LISTA DE TABELAS

Tabela 3.1 – Avaliação das ferramentas quanto ao ajuste organizacional e técnico...28
Tabela 5.1 - Análise dos critérios de avaliação..44

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
BSTQB	Brazilian Software Testing Qualifications Board
CTFL-MAT	Certified Tester Foundation Level - Mobile Application Testing
CTAL-TAE	Certified Tester Advanced Level - Test Automation Engineer
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
ISTQB	International Software Testing Qualifications Board
JDK	Java Development Kit
NPM	Node package manager
SDK	Software Development Kit
USB	Universal Serial Bus
UTF-8	UCS Transformation Format
WEB	World Wide Web
WI-FI	Wireless Fidelity
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO 9
1.1 Objetivos 11
1.1.1 Objetivos Secundários 11
1.2 Estrutura do Texto 11
2 FUNDAMENTAÇÃO TEÓRICA 12
2.1 Desafios dos Testes em Dispositivos Móveis 12
2.1.1 Tipos de Aplicativos 12
2.1.2 Requisitos de Testes Específicos 14
2.2 Testes em Dispositivos Móveis 14
2.3 Automação de Testes em Dispositivos Móveis 15
3 PROPOSTA 20
3.1 Aplicativo Alvo 20
3.1.1 Requisitos para Automação de Testes 24
3.2 Levantamento das Ferramentas para Automação 24
4 FERRAMENTAS SELECIONADAS 29
4.1 Appium (Android e iOS) 29
4.2 XCTest (iOS) 32
4.3 Espresso (Android) 33
4.4 Critérios de Avaliação 33
5 AVALIAÇÃO EXPERIMENTAL 35
5.1 Configurações e Infraestrutura 35
5.2 Arquitetura de Teste 36
5.3 Avaliação da Ferramenta Appium (Android e iOS) 38
5.4 Avaliação da Ferramenta XCTest (iOS) 39
5.5 Avaliação da Ferramenta Espresso (Android) 40
5.6 Discussão 42
6 CONCLUSÃO 45
REFERÊNCIAS 47
APÊNDICE A – CONFIGURAÇÕES DO APPIUM NO MACOS 49
APÊNDICE B – *FRAMEWORKS* E IDE UTILIZADOS COM O APPIUM 55
APÊNDICE C – LINGUAGEM E IDE UTILIZADAS COM O XCTEST 58
APÊNDICE D – LINGUAGEM E IDE UTILIZADAS COM O ESPRESSO 60

1 INTRODUÇÃO

Com o crescimento da demanda por aplicativos de dispositivos móveis (*Smartphone*, *Tablets*, entre outros), é necessário garantir a qualidade do que está sendo desenvolvido. Este objetivo se torna possível através de planejamento e execução dos testes de *software* em aplicativos nativos (sistema operacional Android e/ou iOS), híbridos[1] e/ou baseados em navegadores.

Em aplicações móveis os testes automatizados se demonstram essenciais para atender aos requisitos de demanda elevada e evolução constante, tanto dos sistemas operacionais, quanto dos equipamentos. Os desafios se tornam frequentes pela variedade de versões e dispositivos, cada qual com os seus recursos e necessidades específicas.

Outro agravante comum ocorre quando o projeto utiliza métodos ágeis tendo como base a implementação da automação de testes, que devido a restrições, não abrangem todos os possíveis cenários, logo, testes manuais são necessários para aumentar a cobertura do que está sendo validado (SANTOS; CORREIA, 2015).

Segundo a CTFL-MAT[2] (BSTQB/ISTQB[3], 2019) a estratégia de testes deve levar em consideração questões como:

- compatibilidade (recursos, tamanhos de telas, temperatura, sensores, interrupções, permissões, bateria, entre outros);
- interações (notificações, preferências e diferentes tipos de aplicativos);
- conectividade.

O documento também expõe que o processo de teste pode cobrir instalação, estresse, segurança, performance, usabilidade, entre outros.

Tendo em vista a automação do teste funcional[4] no nível de sistema (baseado na interface do usuário) em aplicativos nativos de dispositivos móveis, existem duas abordagens possíveis que se diferem pela adequação ao sistema operacional, com

---

[1] Funciona como nativo, mas é desenvolvido com tecnologias da *web*
[2] *Certified Tester Foundation Level - Mobile Application Testing*
[3] *Brazilian Software Testing Qualifications Board/International Software Testing Qualifications Board*
[4] Quando o termo teste funcional for utilizado, ele sempre vai se referir ao nível de sistema

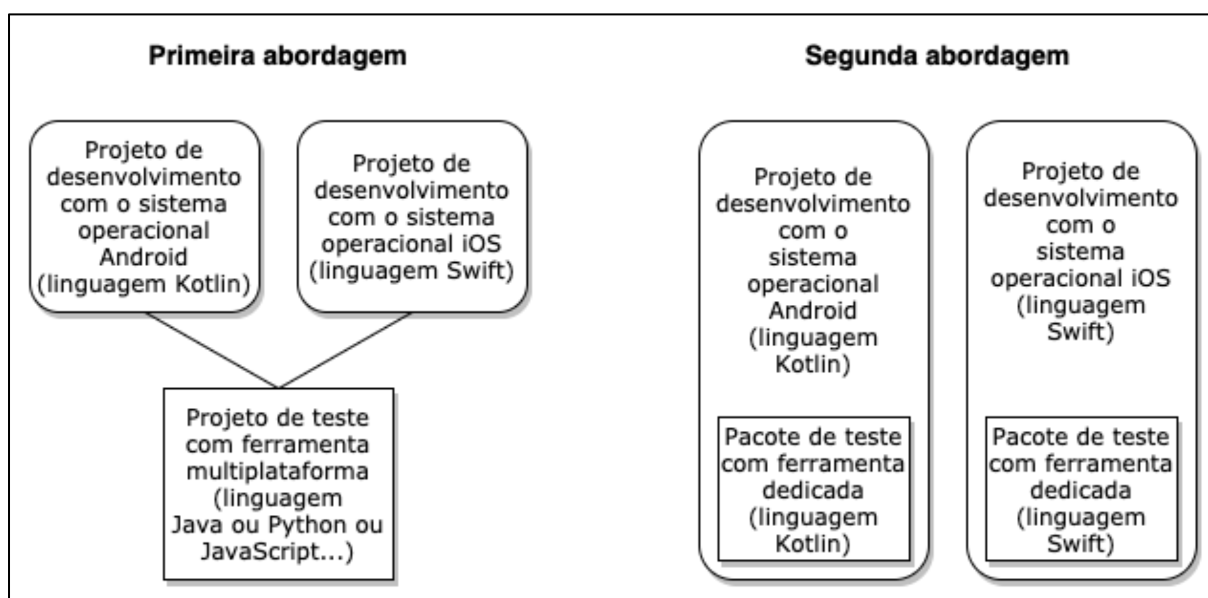
pontos positivos e negativos que influenciam no momento de escolha (BSTQB/ISTQB, 2019).

Na primeira abordagem utilizamos uma ferramenta que atenda diferentes sistemas operacionais dentro do projeto de teste. As desvantagens são limitações específicas na configuração de ambiente e utilização de diferentes linguagens de programação entre os projetos de desenvolvimento e teste.

Já a segunda abordagem tem como foco trabalhar de forma isolada em cada sistema operacional dentro do projeto de desenvolvimento. As limitações deste método refletem em trabalhar em projetos diferentes (por exemplo Android e iOS) com linguagens de programação diferentes (por exemplo Kotlin e Swift) e na provável incompatibilidade entre módulos dependendo da complexidade e arquitetura do projeto de desenvolvimento.

Abaixo é possível observar um diagrama de exemplo das duas abordagens (Figura 1.1).

Figura 1.1 – Diagrama de exemplo das abordagens de automação de testes em aplicativos nativos.



Fonte: O próprio autor.

É impossível garantir a qualidade de todos os produtos desenvolvidos sem o uso da automação de testes, embora ela não seja uma condição suficiente, isso por conta dos desafios já mencionados e prazos curtos de entrega.

Assim, este trabalho propõe o estudo comparativo das duas abordagens de automação de testes mencionadas acima.

1.1 Objetivos

O objetivo principal do presente trabalho é realizar uma análise comparativa de ferramentas mais recentes para automação do teste funcional (baseado na interface do usuário) em aplicativos nativos para dispositivos móveis.

1.1.1 Objetivos Secundários

Os objetivos secundários deste trabalho são:

a. Identificar ferramentas de automação de testes com foco na interação do usuário em aplicativos nativos;
b. Definir critérios para realizar uma análise comparativa;
c. Demonstrar a ferramenta de automação do teste funcional que melhor responda aos requisitos de uma aplicação específica.

1.2 Estrutura do Texto

No Capítulo 2 são discutidos os desafios dos testes, passando pelos tipos de aplicativos e requisitos de testes específicos. Ainda neste capítulo são revisados os principais conceitos sobre a automação de testes em dispositivos móveis.

O Capítulo 3 apresenta a proposta deste trabalho, onde são abordados o aplicativo alvo, passando pelos requisitos e o levantamento das ferramentas para automação de testes.

O Capítulo 4 apresenta as ferramentas selecionadas (Appium, XCTest e Espresso), além disso, são definidos os critérios para análise comparativa experimental.

Já o Capítulo 5 apresenta as configurações e infraestrutura, bem como a arquitetura de teste e avaliação experimental das ferramentas selecionadas. Por fim, é feito uma discussão.

Finalmente, o Capítulo 6 apresenta a conclusão, juntamente com os trabalhos futuros que podem ser realizados para dar continuidade a este trabalho.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são discutidos os desafios dos testes (tipos de aplicativos e requisitos de testes específicos) e os conceitos essenciais sobre automação de testes em dispositivos móveis.

2.1 Desafios dos Testes em Dispositivos Móveis

Dispositivos móveis cada vez são mais utilizados, isso por conta da enorme variedade de aplicativos. Porém existe uma certa complexidade em seu desenvolvimento e, principalmente, sua verificação, para garantir a qualidade (SANTOS; CORREIA, 2015).

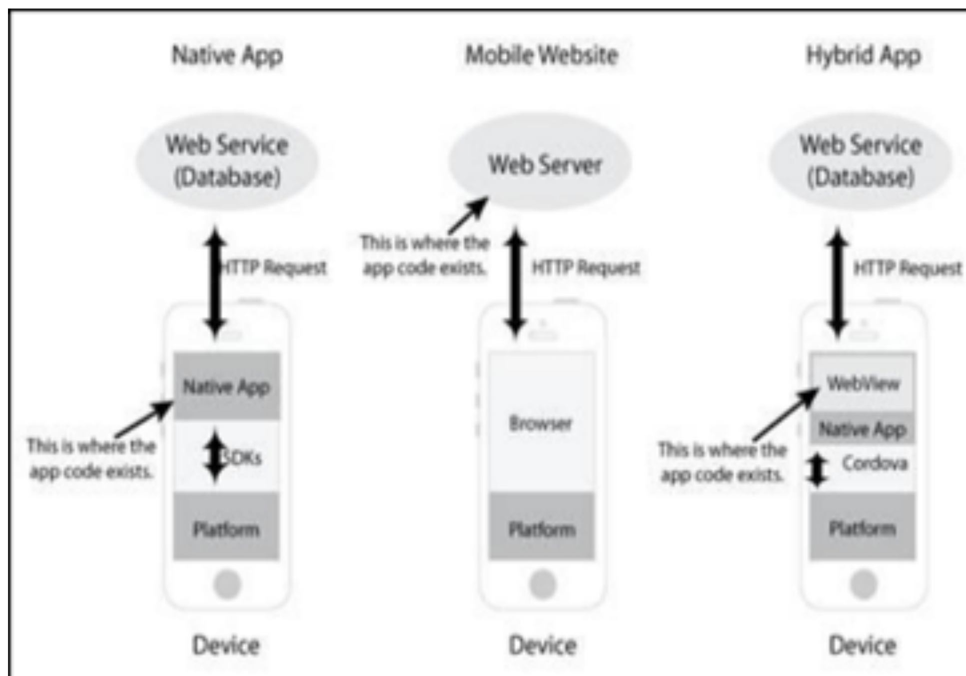
Muitas vezes é necessário obter conhecimentos específicos para superar os desafios (demandas específicas e/ou restrições) tais como: evolução dos dispositivos, variedade de plataformas, aprovações para publicação e o tempo curto para atender às necessidades dos usuários (SANTOS; CORREIA, 2015).

Além disso, características como: desempenho, otimização da interface, rede e usabilidade devem ser cobertas com os testes, porém o tempo é curto quando são utilizados métodos ágeis (SANTOS; CORREIA, 2015).

2.1.1 Tipos de Aplicativos

Existem três tipos de aplicativos para dispositivos móveis: os nativos, os híbridos e os baseados em navegadores. Na Figura 2.1 a seguir é possível observar a arquitetura de cada um (SOUJANYA; RUPA, 2017).

Figura 2.1 – Os três tipos de aplicativos para dispositivos móveis.



Fonte: Soujanya; Rupa (2017).

No caso dos aplicativos nativos, podem ser desenvolvidos utilizando a linguagem de programação Kotlin (KOTLIN, 2021) ou Java para o sistema operacional Android. Já para o sistema operacional iOS é possível utilizar a linguagem de programação Swift (SWIFT, 2021) ou Objective-C. Eles são executados e compilados diretamente nos dispositivos pelo SDK[5]. Já os aplicativos baseados em navegadores são acessados pelo navegador. Um aplicativo híbrido tem a visualização pela *web*[6], porém usa um *wrapper*[7] de aplicativo nativo para se comunicar entre a *web* e a plataforma nativa (SOUJANYA; RUPA, 2017).

---

[5] *Software Development Kit*
[6] *World Wide Web*
[7] Em linguagens de programação, como o JavaScript, um *wrapper* é uma função destinada a chamar uma ou mais funções

2.1.2 Requisitos de Testes Específicos

Embora os testes em dispositivos móveis tenham similaridades com outros tipos de aplicações, eles apresentam alguns requisitos específicos, tais como (SANTOS; CORREIA, 2015, p. 1):

- "interação com outros aplicativos";
- "sensores: acelerômetro, tela sensível ao toque, microfone, câmeras, entre outros";
- "famílias de plataformas de *hardware* e *software*";
- "interfaces de usuário";
- "testes complexos associados à transmissão (rede telefônicas)";
- "consumo de energia do dispositivo / bateria".[8]

2.2 Testes em Dispositivos Móveis

Os testes são uma parte importante do desenvolvimento de aplicativos e podem ser divididos em dois grupos (SOUJANYA; RUPA, 2017):

- Teste de *hardware*, que pode cobrir tamanho de tela, espaço de memória, câmera, Wi-Fi[9], entre outros;
- Teste de *software* ou aplicativo, que pode cobrir as funcionalidades.

Além disso, os tipos de teste que devem ser realizados em aplicativos são: teste de usabilidade, compatibilidade, interface, serviço, performance, entre outros (SOUJANYA; RUPA, 2017).

Outro fator importante que deve ser levado em consideração são as estratégias de testes adotadas, tais como: seleção de dispositivos e o uso de emuladores (SOUJANYA; RUPA, 2017).

De acordo com a CTFL-MAT (BSTQB/ISTQB, 2019), um emulador é definido como:

---

[8] Todos os pontos supracitados são de livre tradução
[9] *Wireless fidelity*

> Emulador móvel: representação virtual de uma plataforma de *hardware*. Por exemplo, o emulador do Android é um *hardware* virtual que executa uma imagem real do sistema operacional Android. A mesma imagem do sistema operacional, se for implantada em um *hardware* ela funcionará.

O uso de emuladores é altamente recomendado por ser mais econômico e por ter uma imensa disponibilidade dos principais dispositivos de mercado. Ainda assim, em uma segunda etapa de testes, devem ser utilizados dispositivos reais para validar características críticas ou que não são possíveis com emuladores (SANTOS; CORREIA, 2015).

Segundo a CTFL-MAT (BSTQB/ISTQB, 2019), um emulador não pode substituir por completo um dispositivo real, pois pode ocorrer um comportamento diferente ao tentar imitá-lo. Além disso, alguns recursos não são suportados, como *multi-touch*, acelerômetro e outros.

Um outro recurso utilizado são os simuladores. Segundo a CTFL-MAT (BSTQB/ISTQB, 2019):

> Um simulador modela o ambiente de tempo de execução, enquanto um emulador modela o *hardware* e utiliza o mesmo ambiente de tempo de execução que o *hardware* físico. As aplicações testadas em um simulador são compiladas em uma versão dedicada, que funciona no simulador, mas não em um dispositivo real. Assim, é independente do sistema operacional real.

Simulador móvel é definido como um ambiente de tempo de execução virtual. Por exemplo, o simulador do iOS finge ser iOS, mas na verdade não é um iOS real.

2.3 Automação de Testes em Dispositivos Móveis

Em projetos de aplicativos para dispositivos móveis, uma recomendação é realizar a automação de testes sempre que possível e que o projeto suporte a execução de testes em dispositivos reais e em emuladores (SANTOS; CORREIA, 2015). A automação de testes se depara com dois desafios, segundo GAO, et al. (2014): 1) a falta de padronização na infraestrutura de teste móvel, linguagens dos scripts e a conectividade entre as ferramentas de teste móvel e as plataformas; 2) a falta de infraestrutura e soluções unificadas que atendam à maioria dos dispositivos móveis.

Em relação ao primeiro desafio, existe uma certa dificuldade para manter a padronização entre emulação, dispositivos físicos, nuvem, entre outros. Além disso, também é complicado manter as mesmas linguagens de programação entre as camadas de testes, bem como a conexão com as plataformas (por exemplo diferentes sistemas operacionais).

Já no segundo desafio, a dificuldade é devido a variabilidade de dispositivos, fabricantes e versões dos sistemas operacionais.

Um dos grandes desafios é montar um ambiente de teste. Devido às atualizações de dispositivos e tecnologias, é necessário um ambiente reutilizável e de baixo custo, bem como que suporte uma grande escala de testes automatizados. Uma possível solução é utilizar testes na nuvem, que oferecem uma enorme quantidade de dispositivos de forma econômica, para atender as diversas necessidades do teste móvel (GAO, et al., 2014).

Segundo a CTAL-TAE[10] (BSTQB/ISTQB, 2016, p. 32) os requisitos para a automação de testes precisam levar em consideração alguns pontos:

- “Qual nível de teste deve ser suportado, por exemplo, nível de componente, nível de integração, nível de sistema”;
- “Que tipo de teste deve ser suportado, por exemplo, teste funcional, teste de conformidade, testes de interoperabilidade”;
- “Que tecnologias o sistema em teste deve ser suportado, por exemplo, para definir a solução de automação de teste em vista da compatibilidade com as tecnologias do sistema em teste”;
- Entre outros.

Para ter sucesso com a automação de testes é necessário ter uma boa estratégia para construir uma estrutura que suporte as necessidades do projeto. Zhifang, Bin e Xiaopeng (2010) ressaltam que é importante levar em consideração os aspectos enumerados abaixo:

1. teste baseado na interface do usuário: esse tipo de teste pode ser mais preciso e abrangente, pois normalmente os aplicativos são projetados

---

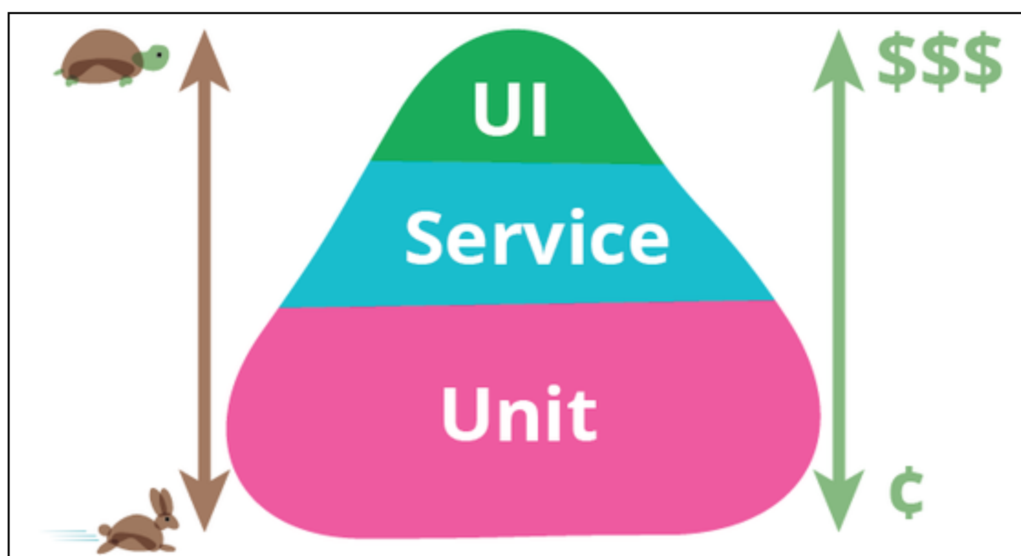
[10] *Certified Tester Advanced Level - Test Automation Engineer*

para dar uma experiência ao usuário, já que a tela fornece *feedback* e carrega diversas informações se for comparado com som e vibração;

2. simulação da operação (toque na tela, clique único, multitoque, deslizar, entre outros);
3. criação de uma arquitetura genérica para implementar os dois itens acima, ou seja, um projeto base com métodos/funções que podem ser reutilizadas por exemplo. No entanto isso é complicado devido à variedade de sistemas operacionais e a complexidade do desenvolvimento.

Para realizar a automação de testes, podemos utilizar o conceito da pirâmide de teste (Figura 2.2) que auxilia sobre a quantidade e os tipos de testes que devem ser feitos (FOWLER, 2012).

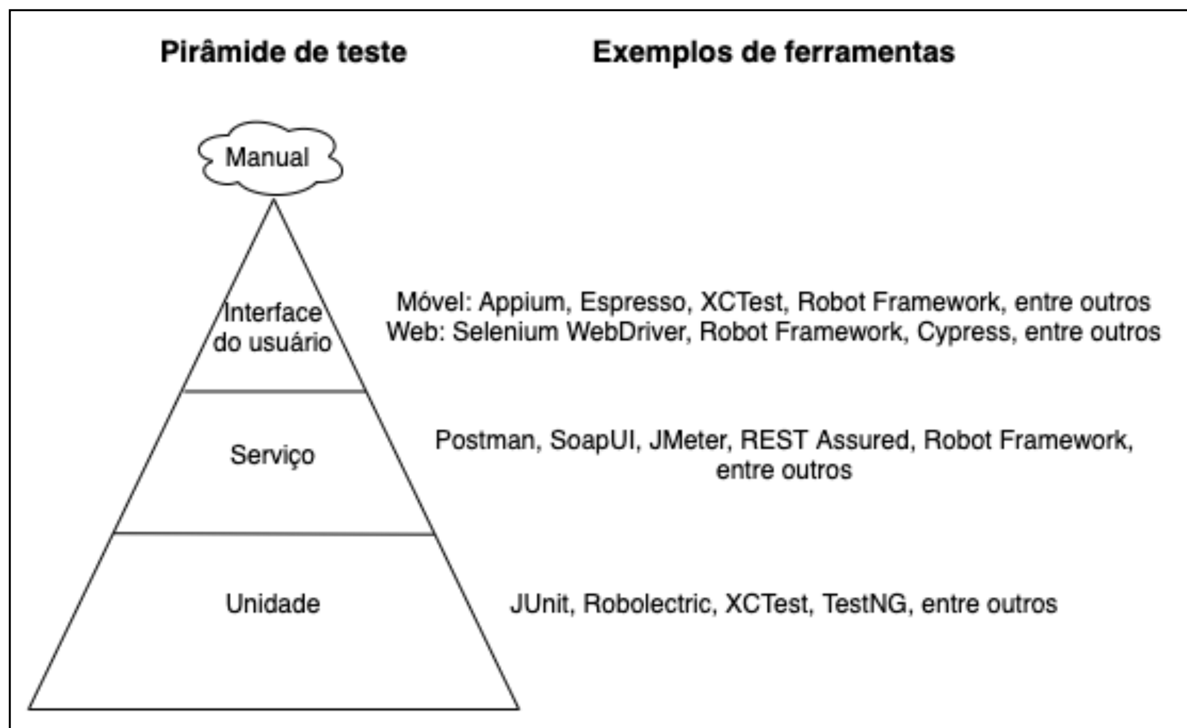
Figura 2.2 – Pirâmide de teste.



Fonte: Fowler (2012).

Para cada camada existem diversas ferramentas (Figura 2.3), por exemplo na camada de unidade, caso a linguagem de programação usada no projeto seja Java, pode ser utilizado o JUnit, já na camada de interface do usuário, caso seja um sistema *web* pode ser usado o Selenium WebDriver ou ser for um aplicativo móvel o Appium.

Figura 2.3 – Exemplos de ferramentas na pirâmide de teste.



Fonte: Adaptado de Fowler (2012).

Esse trabalho tem como aspecto principal a automação do teste funcional (baseado na interação do usuário com a aplicação), visando validar as funcionalidades definidas nos requisitos e observando se o comportamento é o esperado. A estratégia de teste deve incluir ferramentas para facilitar a automação, tais como: Appium, Espresso e XCTest (MACHARLA, 2017). Além disso, irá abordar alguns aspectos da simulação de operação, como por exemplo o clique único e deslizar.

Segundo a CTFL-MAT (BSTQB/ISTQB, 2019, p. 47), os principais recursos para uma arquitetura de teste de aplicativo móvel são:

- “Identificação de objetos”;
- “Operações de objetos”;
- “Relatórios de teste”;
- “Interfaces de programação de aplicativos e recursos extensíveis”;
- “Documentação adequada”;
- “Integrações com outras ferramentas”;
- “Ser Independente das práticas de desenvolvimento de testes”.

Mesmo com as limitações a automação tem diversas vantagens, tais como: evita a repetição de testes; lógica complexa pode ser implementada evitando erros do

teste manual; monitoramento por intermédio de ferramentas; e a redução do custo a longo prazo (ZHIFANG; BIN; XIAOPENG, 2010).

3 PROPOSTA

Como mencionado, a estruturação de um ambiente de teste automatizado, considerando os diferentes tipos de aplicativos e a grande variedade de dispositivos, é um dos maiores desafios para a automação do teste de aplicativos móveis.

Neste trabalho, pretende-se identificar qual ou quais ferramentas de automação da camada de interface do usuário seriam as mais adequadas para o contexto de uma empresa específica. Assim, o primeiro passo é a identificação do sistema em teste e seus requisitos de verificação. A partir disso, foi realizado um levantamento bibliográfico referente às abordagens e ferramentas para automação de teste baseado na interface do usuário em aplicativos nativos para dispositivos móveis. Para isso foram utilizados artigos científicos, certificações de testes de *software*, documentações oficiais das ferramentas e os seus repositórios.

A partir da definição de alguns critérios de avaliação, foi possível identificar as ferramentas que seriam adequadas para uma avaliação experimental, por fim, foi feito um estudo de caso, onde uma estratégia de automação foi implementada, para o aplicativo em teste, usando as ferramentas escolhidas. Com isso foi possível detalhar as vantagens e desafios para o uso de cada uma.

3.1 Aplicativo Alvo

Para execução dos experimentos foi utilizada a funcionalidade de cadastro da instituição financeira Unicred, conhecida como *Onboarding* Digital, uma biblioteca inserida no Unicred *Mobile* (aplicativo nativo que pode ser instalado através das lojas das plataformas Android[11] e iOS[12]). Fica localizado na tela inicial, ou seja, área de acesso livre, permitindo ao futuro cooperado se cadastrar de forma independente.

O aplicativo está sendo bastante utilizado devido à pandemia do coronavírus (covid-19), isso por conta do fechamento de agências em alguns períodos, pessoas

---

[11] https://play.google.com/store/apps/details?id=br.com.unicredmobile
[12] https://apps.apple.com/br/app/unicred-mobile/id955807456

que preferem não ir às agências físicas e até mesmo indicação dos gerentes de relacionamentos.

A Figura 3.1 mostra a tela inicial do Unicred *Mobile*. O botão “Sou Cooperado” direciona para a área de acesso restrito, já o botão “Quero Ser Cooperado” acessa a tela de cadastro do *Onboarding* Digital (Figura 3.2).

Figura 3.1 – Tela inicial do Unicred *Mobile*.

Fonte: O próprio autor.

Figura 3.2 – Primeira tela do *Onboarding* Digital (tela de cadastro)

Fonte: O próprio autor.

A primeira tela do *Onboarding* Digital possui os campos:

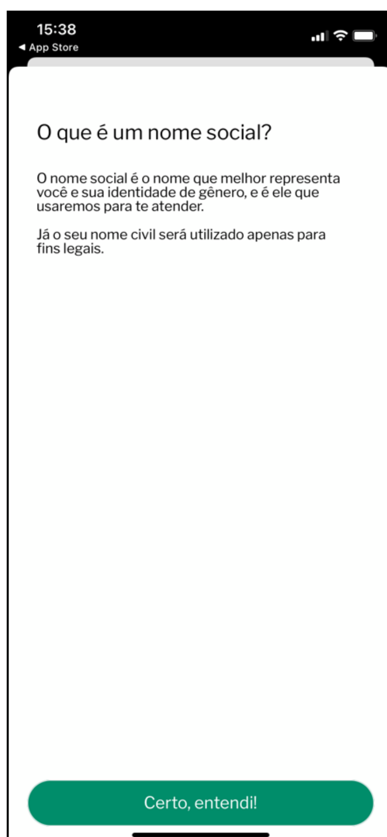
- Número do CPF: campo de preenchimento numérico com máscara obrigatório;
- Nome completo: campo de preenchimento alfabético obrigatório;
- Nome social (opcional): campo de preenchimento alfabético opcional;
- Botão “?”: ao acionar abre uma tela (Figura 3.3) informando o que é um nome social com o botão “Certo, entendi!”;
- Estado de nascimento: campo de seleção obrigatório, ao acionar abre uma lista com todos os estados brasileiros;
- Cidade de nascimento: campo de seleção obrigatório, esse campo fica bloqueado até que seja selecionado um estado de nascimento. Quando habilitado, abre uma lista com todas as cidades do estado de nascimento previamente selecionado;
- Estou casado(a) ou em uma união estável: ao acionar é alterado o texto do botão “Continuar” para “Dados do Cônjuge”, a partir deste momento,

ao acionar o botão “Dados do Cônjuge” é feito o direcionamento para uma tela para preencher os dados do cônjuge;

- Li e concordo com os termos e políticas: botão deslizante (ligado ou desligado) obrigatório para habilitar o botão “Continuar”;
- Ler os termos de uso e políticas de privacidade: ao acionar abre uma tela com todos os termos e condições de uso;
- Botão “Continuar”: esse botão fica desabilitado até que todos os campos obrigatórios sejam preenchidos. Quando habilitado, direciona para tela de contatos ou dados do cônjuge caso tenha sido selecionada a opção correspondente.

A tela de cadastro deste aplicativo é um bom exemplo para avaliação de ferramentas de automação de testes, pois com ela é possível realizar diversos tipos de interações, tais como: toque na tela, digitar em um campo, clique único, deslizar, selecionar opções, validar textos na tela, entre outros.

Figura 3.3 – Tela: O que é um nome social?.



Fonte: O próprio autor.

3.1.1 Requisitos para Automação de Testes

Os requisitos para a automação de testes selecionados de acordo com o sistema alvo foram:

- Formas de interação do usuário (digitação de texto, seleção de opções, visualização de informações, entre outros);
- Simulação de operação (toque na tela, clique único, deslizar, entre outros);
- Utilização de emulador e simulador ao invés de dispositivos reais ou até mesmo alguma solução na nuvem.

A sua escolha foi devido à exigência do aplicativo alvo e a tela selecionada, em relação às diferentes formas de interação do usuário, diferentes tipos e formas de inclusão de dados, além disso, a organização não possui uma ampla variedade de dispositivos reais, então a opção inicial foi utilizar emuladores e simuladores.

Esses requisitos são importantes porque acabam cobrindo em seu contexto a validação dos requisitos (por exemplo um campo de CPF que deve ser obrigatório e aceitar apenas números), o tipo de teste funcional (nível de teste de sistema), o teste de regressão (para quando houver alguma mudança no aplicativo) e o teste de diferentes exibições (para validar os diversos tamanhos de telas com o auxílio do uso de emulador e simulador).

O foco deste trabalho é no nível de teste de sistema, ao invés do teste de unidade ou serviço, já que a organização possui uma cobertura de testes nessas outras camadas da pirâmide de teste. Basicamente será realizado o tipo de teste funcional, ao invés de outros tipos de testes, como por exemplo estresse ou segurança. Como o aplicativo alvo é nativo, a escolha de ferramentas para a arquitetura de teste foi pensada nesses aspectos.

3.2 Levantamento das Ferramentas para Automação

O levantamento realizado das principais ferramentas para automação de testes baseado na interface do usuário em aplicativos nativos (sistema operacional Android e/ou iOS) resultou na lista abaixo:

- Appium: *framework*[13] de código aberto para automação de testes em aplicativos nativos (sistema operacional Android e/ou iOS), híbridos e baseados em navegadores. Com o Appium é possível trabalhar com as linguagens Ruby, Python, Java, JavaScript, PHP, C# e RobotFramework (APPIUM, 2021).
- Espresso: *framework* para automação de testes em aplicativos nativos Android da própria Google. Com o Espresso é possível trabalhar com as linguagens de programação Kotlin e Java (ESPRESSO, 2020).
- XCTest: *framework* para automação de testes em aplicativos nativos iOS da própria Apple. Com o XCTest é possível trabalhar com as linguagens de programação Swift e Objective-C (XCTEST, 2020).
- Barista: *framework* para automação de testes em aplicativos nativos Android construído sobre o Espresso. Ele simplifica algumas tarefas e diminui a quantidade de código, por exemplo o trecho "onView(withId(R.id.button)).perform(click())" escrito com o Espresso ficaria da seguinte forma com o Barista "clickOn(R.id.button)" (BARISTA, 2016).
- Robot *Framework*: *framework* de código aberto para automação de testes *web*, de *Application Programming Interface* (API), *desktop* e em dispositivos móveis. O teste é especificado da forma de cenários, baseado em palavras-chave, ao invés de um código programável. Com o Robot é possível trabalhar com as linguagens de programação Python e Java (ROBOT, 2021).
- Katalon: ferramenta para automação de testes *web*, de API, *desktop* e em dispositivos móveis construída com base no Selenium e Appium. Ela possui o próprio ambiente de desenvolvimento integrado (IDE[14]) (Katalon *Studio*), é possível trabalhar com a linguagem de programação Groovy e tem planos gratuitos e pagos (KATALON, 2020).

[13] Conjunto de classes implementadas em uma linguagem de programação específica, usadas para auxiliar o desenvolvimento de *software*
[14] *Integrated Development Environment*

- TestComplete: ferramenta paga para automação de testes *web*, *desktop* e em dispositivos móveis. Ela possui a própria IDE e é possível trabalhar com as linguagens de programação JavaScript, Python, VBScript, JScript (legado[15]), DelphiScript (legado), C#Script (legado) e C++Script (legado) (TESTCOMPLETE, 2021).
- Xamarin.UITest: *framework* de código aberto para automação de testes em aplicativos Android e iOS (Xamarin.iOS, Xamarin.Android e nativo). Com o Xamarin é possível trabalhar com a linguagem de programação C# (XAMARIN UI TEST, 2021).
- Ranorex: ferramenta paga para automação de testes *web*, *desktop* e em dispositivos móveis. Ela possui a própria IDE (Ranorex *Studio*) e é possível trabalhar com as linguagens de programação C# e VB.NET (RANOREX, 2021).

A definição das ferramentas que seriam utilizadas ao longo do trabalho foi baseada em parâmetros de avaliação que segundo a CTFL-MAT (BSTQB/ISTQB, 2019, p. 48) são divididos em duas categorias:

1. de ajuste organizacional, que de acordo com a CTFL (BSTQB/ISTQB, 2018, p. 90) são:
    a. “avaliação da maturidade da organização, seus pontos fortes e fracos”;
    b. “identificação de oportunidades para um processo de teste melhorado suportado por ferramentas”;
    c. “compreensão das tecnologias usadas pelo(s) objeto(s) de teste, a fim de selecionar uma ferramenta que seja compatível com essa tecnologia”;
    d. “as ferramentas de integração contínua e construção já em uso dentro da organização, a fim de garantir a compatibilidade e integração de ferramentas”;
    e. “avaliação da ferramenta contra requisitos claros e critérios objetivos”;

[15] Legado significa que as linguagens estão desatualizadas e raramente são usadas

f. “consideração sobre se a ferramenta está ou não disponível (e por quanto tempo ficará) em um período de teste gratuito”;
g. “avaliação do fornecedor (incluindo treinamento, suporte e aspectos comerciais) ou suporte para ferramentas não comerciais (p. ex., código aberto)”;
h. “identificação de requisitos internos para treinamento e *mentoring* no uso da ferramenta”;
i. “avaliação das necessidades de treinamento, considerando as habilidades de teste (e automação de testes) daqueles que trabalharão diretamente com a(s) ferramenta(s)”;
j. “consideração de prós e contras de vários modelos de licenciamento (p. ex., comercial ou *open source*)”;
k. “estimativa de uma relação custo-benefício baseada em um caso de negócios concreto (se necessário)”.

2. de ajuste técnico:
    a. “testes de automação de requisitos e complexidades, como o uso de novos recursos como identificação facial, impressão digital e chatbots pelo aplicativo”;
    b. “testes dos requisitos do ambiente, como diversas condições de rede, importações ou criações de dados de teste e virtualização do lado do servidor”;
    c. “recursos de relatório de teste e realimentação”;
    d. “a capacidade do *framework* de gerenciar e conduzir a execução em grande escala localmente ou em um laboratório remoto”;
    e. “integração do *framework* de teste com outras ferramentas utilizadas na organização”;
    f. “suporte e disponibilidade de documentação para *upgrades* atuais e futuros”.

Abaixo é apresentado na Tabela 3.1 as ferramentas agrupadas conforme as duas abordagens mencionadas no Capítulo 1 e os principais parâmetros de avaliação (1.d, 1.i, 2.c, 2.d, 2.e e 2.f) analisados conforme acima descritos, segundo o ajuste organizacional e técnico, buscando a verificação de modo a demonstrar se atende (A) ou não atende (NA) os mesmos, bem como o nível de necessidade, por exemplo baixa ou alta. Esses parâmetros de avaliação foram selecionados devido a abordar assuntos

relacionados à organização, treinamento, relatório, execução e documentação. Os mesmos são importantes por conta do contexto da empresa e do aplicativo alvo.

Tabela 3.1 – Avaliação das ferramentas quanto ao ajuste organizacional e técnico.

FERRAMENTAS	PARÂMETROS DE AVALIAÇÃO					
	1.d	1.i	2.c	2.d	2.e	2.f
Appium	A	Baixa	A	A	A	A
Robot	NA	Alta	A	A	NA	A
Katalon	NA	Alta	A	A	NA	A
TestComplete	NA	Alta	A	A	NA	A
Xamarin	NA	Alta	NA	NA	NA	A
Ranorex	NA	Alta	A	A	NA	A
Espresso	A	Alta	A	A	A	A
XCTest	A	Alta	A	A	A	A
Barista	NA	Alta	NA	A	NA	A

Fonte: O próprio autor.

Após a avaliação dos parâmetros, foram pré-selecionadas as ferramentas Appium (suporta o Android e iOS no mesmo projeto), Espresso (nativo Android) e XCTest (nativo iOS) para uma avaliação mais detalhada. Para utilizar o Appium foi selecionada a linguagem de programação Java, pois a organização normalmente usa Java ou Python, além disso, essa escolha mantém a *stack*[16] de tecnologias da camada de serviço.

As ferramentas acima foram escolhidas, porque conforme a Tabela 3.1 são as que atende o maior número de parâmetros de avaliação, sendo que um dos pontos mais importantes é relacionado à organização (no caso 2 parâmetros de avaliação), seguido de pontos que envolvem documentação, execução, relatório, entre outros.

[16] Conjunto de tecnologias que são utilizadas em um projeto

4 FERRAMENTAS SELECIONADAS

Considerando a proposta que foi definida no capítulo anterior, foi necessário selecionar ferramentas que atendam as necessidades da automação de testes em um aplicativo nativo de cadastro, focando na camada de interface do usuário da pirâmide de teste, em um contexto de uma empresa específica.

Abaixo são apresentadas as ferramentas para as duas abordagens mencionadas na introdução, o Appium que atende a primeira abordagem e o XCTest e Espresso que correspondem a segunda abordagem, respectivamente iOS e Android. Além disso, são definidos os critérios para a análise comparativa experimental.

4.1 Appium (Android e iOS)

O Appium é um *framework* de código aberto para automação de testes em aplicativos nativos (sistema operacional Android e/ou iOS), híbridos ou baseados em navegadores. Com o Appium é possível trabalhar com as linguagens Ruby, Python, Java, JavaScript, PHP, C# e RobotFramework (APPIUM, 2021).

Um grande diferencial do Appium, é a possibilidade de suportar as plataformas Android e iOS em um mesmo projeto.

A filosofia do Appium possui quatro princípios (APPIUM, 2021):

- “você não deve ter que recompilar seu aplicativo ou modificá-lo de qualquer forma para automatizá-lo”;
- “você não deve ficar preso a uma linguagem ou *framework* específico para escrever e executar seus testes”;
- “um *framework* de automação em dispositivos móveis não deve reinventar a roda quando se trata de API de automação”;
- “um *framework* de automação em dispositivos móveis deve ser de código aberto, em espírito e prática, bem como no nome”.[17]

---

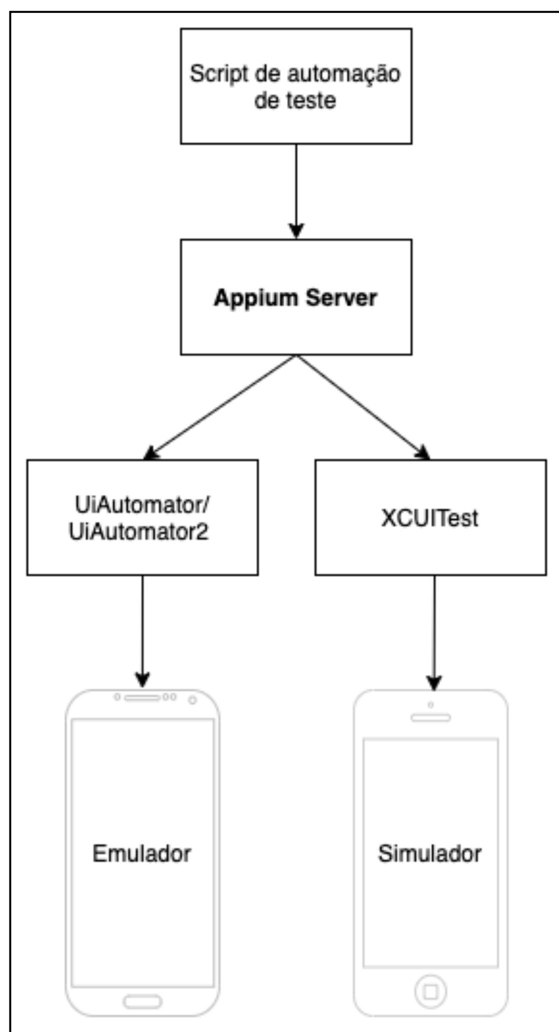
[17] Todos os pontos supracitados são de livre tradução

Para atender ao primeiro princípio, o Appium utiliza os *frameworks* de automação fornecidos pelas próprias plataformas (por exemplo o XCUITest da Apple e o UiAutomator/UiAutomator2 do Google). Isso evita a compilação de algum código ou *framework* específico do Appium ou de terceiros.

O segundo e terceiro princípios são atendidos através da utilização do Selenium WebDriver (*framework* de automação de testes para navegadores *web*). Com o mesmo é possível trabalhar com as principais linguagens de programação. Além disso, não foi necessário fazer um *framework* totalmente diferente e sim apenas estender com funcionalidades extras para automação de testes em dispositivos móveis.

Abaixo é possível observar um diagrama de exemplo da arquitetura do Appium (Figura 4.1).

Figura 4.1 – Arquitetura do Appium.



Fonte: O próprio autor.

O conceito do Appium é de um servidor *web* que expõe uma API REST (API REST, 2021). Assim ele recebe solicitações de um cliente e executa comandos em um dispositivo móvel como uma resposta HTTP[18] que representa esses comandos. Além disso, é possível configurar as *capabilities*[19], que são um conjunto de chaves e valores que informa o Appium sobre qual sessão deve ser iniciada, ou seja, o nome da plataforma (Android e iOS) (APPIUM, 2021).

O Appium possui dois recursos importantes (APPIUM, 2021):

- Appium *Server*: servidor escrito em Node.js que pode ser instalado por meio de comandos NPM[20];
- Appium *Desktop*: interface gráfica do usuário que pode ser instalada em qualquer sistema operacional (Windows, macOS e Linux) e já vem com todos os recursos do Appium *Server*, mas inclui a funcionalidade de inspeção de elementos, que possibilita realizar as interações nos testes.

Para utilizar o Appium, após realizar as configurações de ambiente é necessário selecionar uma linguagem de programação (neste trabalho foi utilizado o Java como mencionado na seção 3.2). Feito essa seleção, vai ser necessário escolher um *framework* de automação de testes de unidade para criação e execução dos testes, uma ferramenta de gerenciamento de dependências e uma IDE ou editor de código.

Neste trabalho, foi utilizado respectivamente o TestNG devido a algumas vantagens, como por exemplo a criação de suítes para cada plataforma (Android e iOS); Gradle devido a possibilidade de codificação com alguma linguagem de programação; e o Intellij que fornece por exemplo preenchimento de código instantâneo, suporte às várias linguagens de programação, *frameworks* e sugestões de nome de arquivos. Além disso, foi utilizado o Hamcrest para facilitar as asserções dos testes.

Os *frameworks* e IDE citados acima podem ser visualizados com mais detalhes no apêndice B.

---

[18] *Hypertext Transfer Protocol*
[19] Foi utilizado o termo original por falta de uma tradução mais adequada
[20] *Node package manager*

4.2 XCTest (iOS)

XCTest é um *framework* da Apple para automação de testes de unidade, desempenho e de interface do usuário em aplicativos nativos iOS. Com o XCTest é possível trabalhar com as linguagens de programação Swift e Objective-C (XCTEST, 2020).

Segundo a documentação do XCTest (XCTEST, 2020):

> Os testes afirmam que 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.[21]

Ele possui a biblioteca de testes de interface do usuário que realiza a validação que o comportamento seja correto, quando determinadas ações esperadas são realizadas. Uma das principais classes utilizadas para esse tipo de teste é a XCTestCase, que serve para definir os casos de teste e métodos de teste. Um caso de teste é um grupo de métodos de teste relacionados a configurações antes e depois da execução dos testes. A classe XCTestCase, tem acesso a classe XCUIApplication, que é uma extensão da classe XCUIElement e tem acesso a métodos de interações como toque na tela, digitar em um campo, clique único, deslizar, entre outros.

Para utilizar o XCTest, é necessário ter o código fonte da aplicação, selecionar uma linguagem de programação (neste trabalho foi utilizado o Swift devido ao projeto de desenvolvimento) e ter a IDE Xcode.

A linguagem de programação e a IDE citadas acima podem ser visualizadas com mais detalhes no apêndice C.

---

[21] Livre tradução

4.3 Espresso (Android)

O Espresso é um *framework* do sistema operacional Android para automação de testes de interface do usuário em aplicativos nativos Android. Com o Espresso é possível trabalhar com as linguagens de programação Kotlin e Java (ESPRESSO, 2020).

Segundo a documentação do Espresso (ESPRESSO, 2020):

> Os testes do Espresso são executados de maneira idealmente rápida! Ele permite ignorar suas esperas, sincronizações, suspensões e pesquisas enquanto faz manipulações e declarações na interface do usuário do aplicativo quando ela está em repouso.

Um dos principais métodos do Espresso é o onView(). Nele é passado o identificador do elemento e, após isso, é possível acessar o método perform(), que realiza a chamada dos métodos de interações como clicar, digitar em um campo, validar textos na tela, entre outros (ESPRESSO, 2020).

Para utilizar o Espresso, é necessário ter o código fonte da aplicação, selecionar uma linguagem de programação (neste trabalho foi utilizado o Kotlin devido ao projeto de desenvolvimento) e ter a IDE Android *Studio*.

A linguagem de programação e a IDE citadas acima podem ser visualizadas com mais detalhes no apêndice D.

4.4 Critérios de Avaliação

Para a análise comparativa das ferramentas selecionadas, foram definidos os critérios de avaliação abaixo:

- Configuração do ambiente: quantidade de dependências e nível de dificuldade;
- Facilidade de uso das ferramentas: dependências para execução dos testes automatizados e acesso aos aplicativos nos emuladores e/ou simuladores;
- Quantidade de código: quantidade de linhas de código para automatizar alguma interação;

- Interações: possibilidade de realizar as interações planejadas no aplicativo alvo;
- Mapeamento dos elementos: como realizar a identificação dos elementos (ferramenta específica, arquivos no projeto ou IDE);
- Desempenho das ferramentas: tempo de execução dos testes automatizados;
- Disponibilidade de documentação e suporte das ferramentas (oficial e outras fontes): facilidade de encontrar a documentação e se a mesma é simples, razoável ou abrangente.

Cada um desses critérios, será avaliado com alguns valores específicos, por exemplo simples, razoável e complexo para o critério facilidade de uso das ferramentas. Onde simples, por exemplo, significa que uma ferramenta é completa e consegue executar os testes automatizados de forma independente, já complexo refere-se à exigência de outras ferramentas para tal execução, bem como configurações específicas de acesso as plataformas Android e iOS.

Espera-se que uma ferramenta seja excelente em tudo, porém isso normalmente não é possível, devido a enorme quantidade de critérios, cada qual com suas necessidades específicas.

5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo são apresentadas as configurações e infraestrutura, a arquitetura de teste utilizada, bem como a avaliação experimental das ferramentas selecionadas, com base nas duas abordagens específicas de aplicativos nativos mencionadas no Capítulo 1, onde o Appium representa a primeira abordagem e o XCTest juntamente com o Espresso, correspondem a segunda abordagem.

5.1 Configurações e Infraestrutura

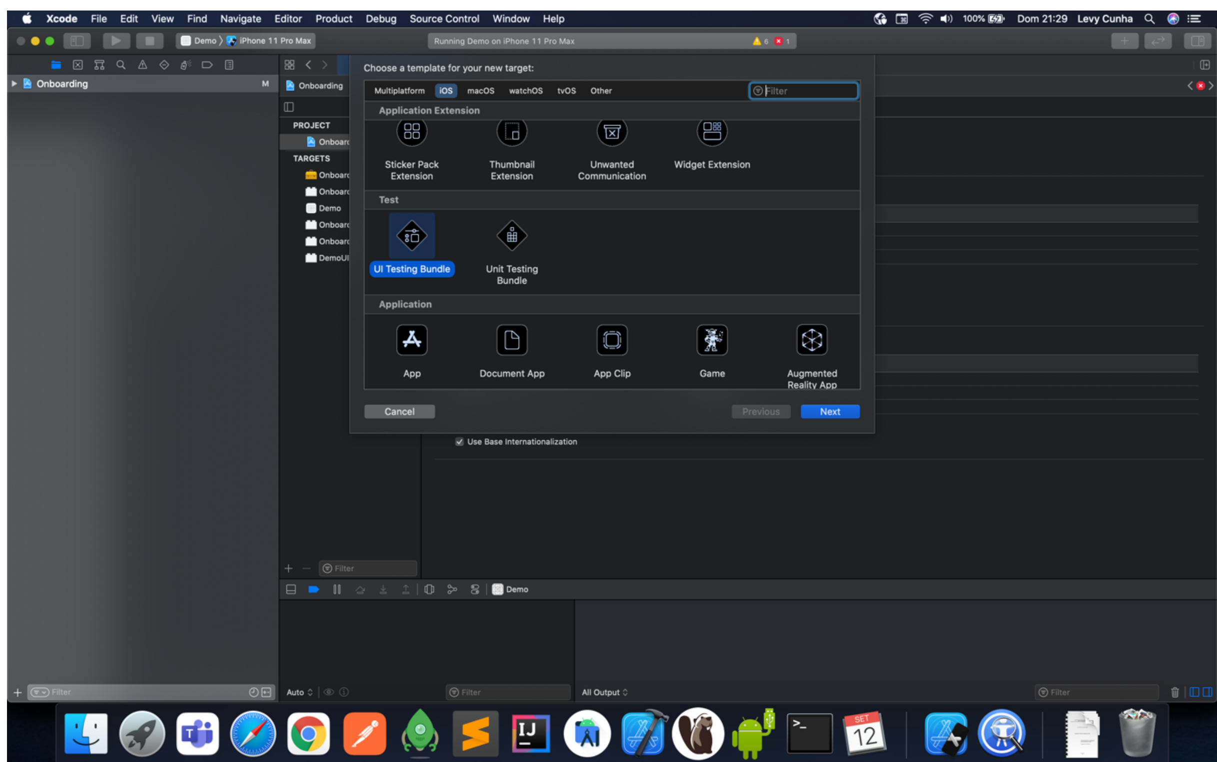
Antes de iniciar o desenvolvimento dos experimentos é necessário baixar e instalar algumas dependências, além disso, devem ser feitas algumas configurações específicas para cada *framework*. Abaixo é possível observar um breve resumo de cada *framework.*

Appium: é necessário realizar uma extensa configuração do ambiente (pode ser visualizada com mais detalhes no apêndice A). Essa configuração, em um primeiro momento, consiste em instalar o Android *Studio* e o Xcode (para isso é necessário ter um Mac), bem como criar um emulador no Android *Studio*. Após isso, é necessário fazer outras instalações, tais como: Node.js, Java JDK[22], Appium *Server*, Appium *Desktop* e o Appium *Doctor*. Com o Appium *doctor* instalado vai ser possível verificar se os requisitos da máquina estão corretos, nesse momento talvez seja necessário configurar as variáveis de ambiente. No macOS por exemplo, devemos inserir as variáveis de ambiente no arquivo “bash_profile” e reiniciar a máquina (se isso não for feito as variáveis não são aplicadas). Uma outra ferramenta importante é o Appium *Desktop*. Por intermédio dele é possível realizar o mapeamento dos elementos para fazer as interações. Para configurar basta inserir as variáveis de ambiente e as *capabilities* (localização do aplicativo, nome da plataforma, versão da plataforma, nome do dispositivo, entre outros).

[22] Java *Development Kit*

XCTest: é necessário instalar o Xcode. Após isso, basta baixar o projeto de desenvolvimento e adicionar o modelo de pacote de teste de interface do usuário (Figura 5.1). Uma ferramenta de desenvolvimento do Xcode importante é o *Accessibility Inspector*. Por intermédio dele é possível realizar o mapeamento dos elementos (*label*, *value* e *identifier*).

Figura 5.1 – Pacote de teste de interface do usuário no XCTest.



Fonte: O próprio autor.

Espresso: é necessário instalar o Android *Studio* e o Java JDK, bem como criar um emulador.

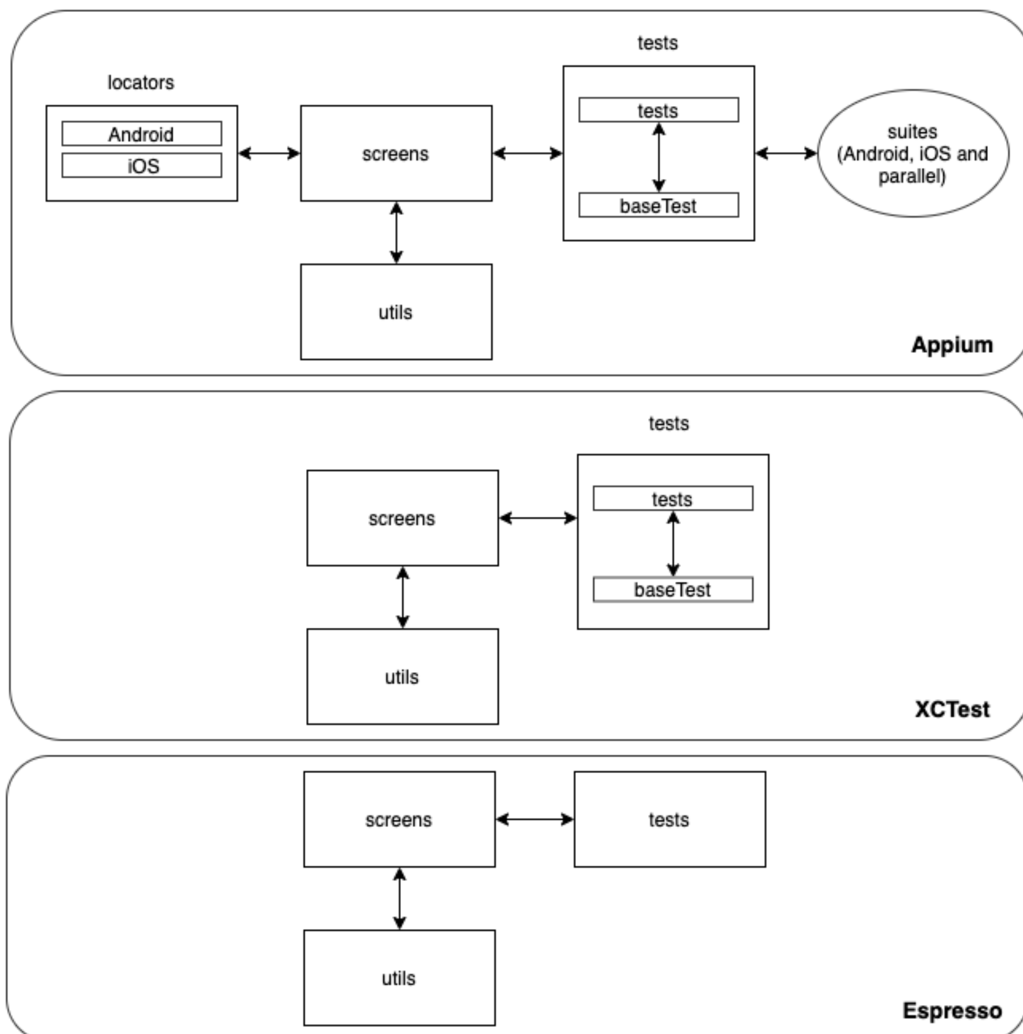
5.2 Arquitetura de Teste

Como este trabalho tem o foco na automação de testes da interface do usuário, o primeiro passo que deve ser feito, independente da ferramenta é o mapeamento de elementos. Após isso, é o momento de iniciar a interação com esses elementos (por exemplo digitar em um campo), para isto foi criado métodos/funções. Por fim, são

realizados os testes, onde cada um pode ter diversos métodos/funções de interações, bem como as validações (mais conhecido como asserções).

A Figura 5.2 mostra o diagrama da arquitetura de teste de cada ferramenta, onde os elementos e as interações ficam localizados nas *screens* (para os elementos do Appium foi criado uma divisão a mais por conta do Android e iOS), as mesmas se comunicam com os *utils* que possuem alguns facilitadores. A última parte da arquitetura seria a de *tests*, onde as *screens* são chamadas. No caso do Appium e XCTest foi criado uma baseTest com algumas dependências para antes e depois dos testes. Para executar os testes com o Appium foram criadas as suítes Android, iOS e paralela com as duas plataformas juntas que serão executadas simultaneamente.

Figura 5.2 – Diagrama da arquitetura de teste.



Fonte: O próprio autor.

Para o desenvolvimento dos experimentos, foi utilizado o padrão de projeto *Page Objects*. O mesmo realiza a organização dos testes, reduzindo a quantidade de código duplicado e facilitando a manutenção (PAGE OBJECTS, 2015). O Appium por exemplo possui recursos do *Page Objects* e do Selenium PageFactory por intermédio do Appium Java *Client* (APPIUM JAVA CLIENT, 2016).

Além disso, foi utilizada a plataforma GitHub para armazenar os projetos de Appium[23], XCTest[24] e Espresso[25], sempre com o uso de fluxo de criação de *branch* e abertura de *pull request* para revisões de código.

5.3 Avaliação da Ferramenta Appium (Android e iOS)

Com o ambiente totalmente configurado, pode-se iniciar a implementação dos testes automatizados de interface do usuário. Com o Appium é possível interagir com os elementos através de diversas estratégias de seletores, incluindo o Xpath, que não é recomendado utilizar, porém alguns elementos não possuem *Accessibility ID*, *Class name*, *ID*, *Name*, entre outros.

Após a identificação dos elementos por intermédio do Appium *Desktop*, foram realizadas diversas interações no aplicativo *Onboarding* Digital, tais como:

1. Clicar no botão “Quero ser Cooperado” para acessar a primeira tela do *Onboarding* Digital;
2. Preencher o campo de CPF;
3. Preencher o campo nome completo;
4. Fechar o teclado;
5. Selecionar o estado de nascimento;
6. Selecionar a cidade de nascimento;
7. Deslizar para cima;
8. Clicar em “Li e concordo com os termos e políticas";
9. Clicar no botão “Continuar” para ser direcionado para próxima tela;
10. Validar o texto “Contatos” da próxima tela.

---

[23] https://github.com/LevyMarcelo/appium-mobile-test
[24] https://github.com/LevyMarcelo/xctest-mobile-test
[25] https://github.com/LevyMarcelo/espresso-mobile-test

O intuito dessas interações foi de simular parte do fluxo de cadastro do *Onboarding* Digital, com o preenchimento de todos os campos obrigatórios da primeira tela.

5.4 Avaliação da Ferramenta XCTest (iOS)

Com as configurações feitas, pode-se iniciar a implementação dos testes automatizados de interface do usuário, dentro da pasta que foi criada (por exemplo OnboardingUITests). Diferente do Appium que atende diversas linguagens de programação, o XCTest atende apenas o Swift (no momento está sendo mais utilizado para esse tipo de desenvolvimento) e Objective-C (linguagem anterior ao Swift), sendo que é possível testar apenas a plataforma iOS.

A interação com os elementos pode ser feita através do seu *label, value* e *identifier*. Uma enorme dificuldade encontrada foi no momento de interagir com os elementos quando os mesmos não possuem nenhuma forma de inspeção. Uma estratégia aplicada foi acrescentar *value* nos campos CPF e nome completo, já que em outros campos isso não foi possível pela falta de conhecimento e tempo. Por conta disso, as interações foram diferentes das realizadas com o Appium.

Após a identificação dos elementos por intermédio do *Accessibility Inspector*, foram realizadas algumas interações no aplicativo *Onboarding* Digital, tais como:

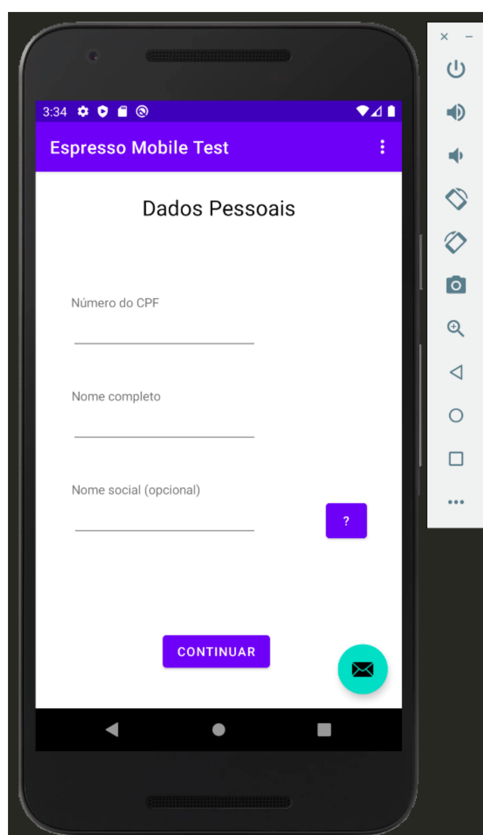
1. Preencher o campo de CPF (foi iniciado por esse campo ao invés do botão “Quero ser Cooperado”, porque foi utilizado o projeto isolado ao invés do projeto Unicred *Mobile* (recentemente o projeto isolado foi acoplado dentro dele), isso para facilitar a implementação dos testes);
2. Preencher o campo nome completo;
3. Clicar em “?” para acessar a tela com informações sobre o nome social;
4. Validar o texto “O que é um nome social?”;
5. Clicar no botão “Certo, entendi!”;
6. Deslizar para cima;
7. Validar o texto do botão “Continuar”.

O intuito dessas interações foi de simular uma pequena parte da primeira tela do *Onboarding* Digital.

5.5 Avaliação da Ferramenta Espresso (Android)

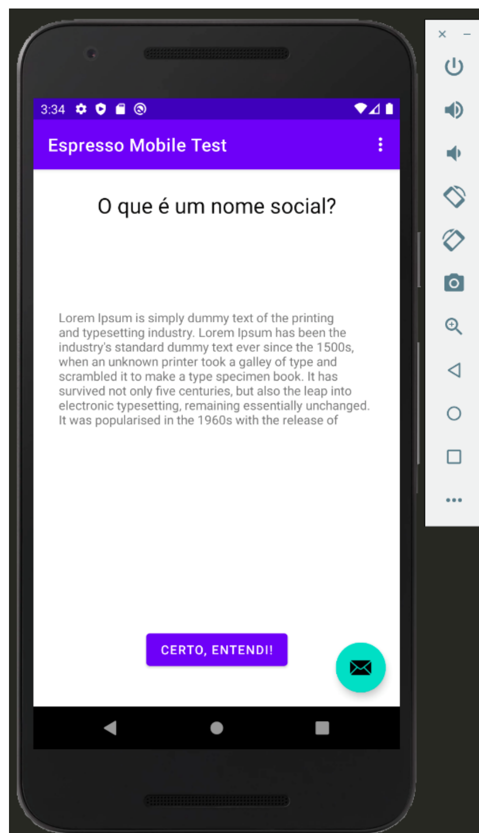
Com o ambiente configurado, basta baixar o projeto de desenvolvimento e criar as classes de testes dentro de “app/src/androidTest/java/<padrão da organização>", porém o projeto *Onboarding* Digital isolado não funciona mais devido às dependências de bibliotecas que estavam no Nexus e foram excluídas, já o projeto Unicred *Mobile* (que agora tem o *Onboarding* Digital dentro) tem diversas incompatibilidades e a falta de configurações para suporte desse tipo de teste. Uma estratégia aplicada para realizar os experimentos foi criar um aplicativo bem básico com alguns campos simbolizando o *Onboarding* Digital (Figura 5.3 e Figura 5.4).

Figura 5.3 – Primeira tela do aplicativo desenvolvido.



Fonte: O próprio autor.

Figura 5.4 – Segunda tela do aplicativo desenvolvido.



Fonte: O próprio autor.

A Inspeção dos elementos foi feita por intermédio dos arquivos xml[26] (por exemplo "fragment_first.xml"), tanto na parte do código, quanto na parte de design.

Diferente do Appium e o XCTest, o Espresso atende apenas as linguagens de programação Kotlin (no momento está sendo mais utilizado para esse tipo de desenvolvimento) e Java, sendo que é possível testar apenas a plataforma Android.

A interação com os elementos pode ser feita através do seu *id*, o mesmo pode ser inserido no momento do desenvolvimento nos arquivos xml (por exemplo "fragment_first.xml"), facilitando assim a implementação dos testes automatizados de interface do usuário.

[26] *eXtensible Markup Language*

Após a identificação dos elementos, foram realizadas algumas interações similares ao que foi feito com o XCTest no aplicativo criado (similar ao *Onboarding* Digital), tais como:

1. Preencher o campo de CPF;
2. Preencher o campo nome completo;
3. Fechar o teclado;
4. Clicar em “?” para acessar a tela com informações sobre o nome social;
5. Validar o texto “O que é um nome social?”;
6. Clicar no botão “Certo, entendi!”;
7. Validar o texto do botão “Continuar”.

O intuito dessas interações foi de simular uma pequena parte da primeira tela do *Onboarding* Digital por intermédio do aplicativo criado.

5.6 Discussão

Abaixo encontra-se a análise dos critérios de avaliação para cada ferramenta selecionada (Appium, XCTest e Espresso).

Configuração de Ambiente: cada ferramenta tem suas particularidades em relação as configurações, o XCTest e o Espresso são simples devido ao número reduzido de dependências. Basicamente é necessário instalar a suas IDEs, no caso do Espresso, também precisa ter o Java JDK, bem como criar um emulador. Já o Appium é complexo devido a diversas dependências, como por exemplo o Android *Studio*, Xcode, emulador, Node.js, Java JDK, Appium *Server*, variáveis de ambiente, entre outros (maiores detalhes podem ser visualizados na seção 5.1).

Facilidade de uso das ferramentas: a utilização do XCTest é simples por se tratar de uma ferramenta completa, a mesma não possui dependências de outras ferramentas, sendo possível por exemplo escrever e executar os testes. No caso do Espresso é razoável devido a algumas dependências, por exemplo o JUnit para execução dos testes. Já a utilização do Appium é complexa, devido a diversas dependências de outras ferramentas e das configurações para acessar os aplicativos no emulador e simulador de ambas as plataformas (respectivamente Android e iOS).

Quantidade de código: no XCTest e Espresso a quantidade de linhas de código para realizar alguma interação é reduzido em muitas partes do projeto, em relação ao

Appium que é extenso, isso pelo fato dos localizadores dos elementos e do acesso as plataformas Android e iOS.

Interações: com o Appium foi possível realizar todas as interações planejadas, diferente do XCTest que foi realizado algumas interações conforme mencionadas na seção 5.4, já com o Espresso não foi executado nenhuma interação com o aplicativo alvo original, conforme a seção anterior.

Mapeamento dos elementos: com o XCTest e Espresso a identificação dos elementos é simples, isso porque respectivamente um fornece uma ferramenta na própria IDE (Xcode) e no outro é possível visualizar em arquivos específicos (código ou design). Já com o Appium é razoável devido a ter que fazer as configurações no Appium *Desktop*.

Desempenho das ferramentas: no XCTest e Espresso as execuções dos testes são mais rápidas que com o Appium, isso porque o mesmo precisa se comunicar com as ferramentas nativas antes de acessar o aplicativo de ambas as plataformas (Android e iOS).

Disponibilidade de documentação e suporte das ferramentas (oficial e outras fontes): por intermédio de buscas na *internet* pela palavra-chave “Appium”, foi possível identificar a sua documentação oficial que é bem abrangente, a mesma é dividida por seções de introdução, como iniciar, história, documentação com todos os seus recursos e exemplos, entre outros. Além disso, foi exibido blogs (por exemplo Medium), GitHub, vídeos, cursos e vários sites falando sobre a ferramenta (por exemplo BrowserStack). Também foram realizadas buscas pelas palavras-chave “Espresso Android” (para se diferenciar do café), onde foi possível identificar a sua documentação oficial que está dentro da seção de teste do Android, a mesma possui algumas seções, como a visão geral, configuração, recursos da ferramenta com exemplos, entre outros. Além disso, foi exibido cursos, blogs (por exemplo Medium) e alguns sites falando sobre a ferramenta (por exemplo BrowserStack). Por fim, foram realizadas buscas pela palavra-chave “XCTest”, onde foi possível identificar a sua documentação oficial que está dentro do desenvolvimento da Apple, a mesma possui algumas seções, sendo que a mais importante para o trabalho é a de teste de interface do usuário, dentro da mesma tem todos os recursos para esse tipo de teste, mas sem exemplos. Além disso, foi exibido alguns sites falando sobre a ferramenta (por exemplo AWS), GitHub e vídeos.

A Tabela 5.1 apresenta um resumo da análise dos critérios de avaliação.

Tabela 5.1 - Análise dos critérios de avaliação.

REQUISITOS	APPIUM	XCTEST	ESPRESSO
Configuração de ambiente	Complexa	Simples	Simples
Facilidade de uso das ferramentas	Complexa	Simples	Razoável
Quantidade de código	Extenso	Reduzido	Reduzido
Interações	Todas realizadas	Algumas realizadas	Nenhuma realizada
Mapeamento dos elementos	Razoável	Simples	Simples
Desempenho das ferramentas	Razoável	Rápido	Rápido
Disponibilidade de documentação e suporte das ferramentas (oficial e outras fontes)	Abrangente	Simples	Razoável

Fonte: O próprio autor.

6 CONCLUSÃO

Este trabalho propôs uma análise comparativa de ferramentas para automação do teste funcional, no qual o foco principal trata-se da simulação de interação do usuário com o aplicativo nativo, sendo ele, sistema operacional Android ou iOS para dispositivos móveis. Neste estudo, foram consideradas duas abordagens mencionadas na introdução, que se diferem pela adequação ao sistema operacional.

Devido à variedade de dispositivos e versões aliado aos prazos curtos de entrega, esse tipo de automação torna-se essencial para garantir a qualidade do produto desenvolvido, embora não seja uma condição autossuficiente. A automação possui seus próprios desafios, por exemplo a conectividade do teste com as plataformas e falta de uma infraestrutura que atenda todos os dispositivos móveis.

Embasado no conceito da pirâmide de teste e cobertura pré-existente dentro da organização (unidade e serviço), a tomada de decisão focou-se na camada superior (interface do usuário).

Inicialmente, para alcançar os objetivos deste estudo, foi selecionado o aplicativo denominado *Onboarding* Digital (módulo de cadastro do Unicred *Mobile*), que pertence à instituição financeira Unicred. Com este, foram realizados diversos tipos de interações, abordados na avaliação experimental. Os requisitos que definiram os itens para automação de testes, baseiam-se na forma de interação do usuário e da simulação de operação.

Em um segundo momento, foram levantadas as ferramentas disponíveis no mercado e suas especificidades. A definição de quais fariam parte dos experimentos, aplicou-se os parâmetros de avaliação baseados na camada de interface do usuário da pirâmide de teste.

Partindo das definições citadas acima, foram pré-selecionadas as ferramentas para avaliação experimental, sendo elas, Appium que atende tanto Android, quanto iOS, referente a primeira abordagem e o XCTest (iOS) em conjunto com o Espresso (Android) correspondente a segunda abordagem, também foram definidos os critérios de avaliação para a análise comparativa.

Para dar início a avaliação experimental, efetuam-se as configurações e preparação da infraestrutura dos testes.

Nesse momento foi possível observar que o Appium tem diversas dependências, o que torna sua configuração complexa e a infraestrutura deficitária quando comparada aos demais. Apesar da complexidade, com ele foi possível realizar todas as interações planejadas na tela de cadastro do aplicativo alvo.

Já com o XCTest, após uma pequena configuração se inicia a implementação dos testes, entretanto, a ferramenta demonstrou dificuldade no mapeamento dos elementos, pois quando eles não possuem nenhuma forma de inspeção torna-se necessário acrescentar valores para prosseguir, outro ponto negativo constatado, foi a parcialidade na execução das interações planejadas.

O Espresso apesar de possuir configurações simples, devido a suas dependências, impossibilitou a realização das interações planejadas no aplicativo alvo, necessitando a criação de um aplicativo similar.

Partindo dos experimentos realizados, foi possível elaborar a análise dos critérios de avaliação, na qual o *framework* Appium demonstra atender aos requisitos do aplicativo alvo de forma mais completa, mesmo com suas configurações complexas e da necessidade de construção da infraestrutura.

Como trabalhos futuros propõe-se a configuração dos projetos para que suportem execuções, tanto em dispositivos reais, quanto em serviços de *device farm* em nuvem, bem como a criação de *pipelines* com ferramenta de integração contínua e implementação de relatórios das execuções dos testes.

REFERÊNCIAS

ANDROID STUDIO. **Download Android Studio and SDK tools | Android Developers**. Disponível em: <https://developer.android.com/studio>. Acesso em: 6 jul. 2021.

API REST. **O que é API REST? - Red Hat**. Disponível em: <https://www.redhat.com/pt-br/topics/api/what-is-a-rest-api>. Acesso em: jul. 2021.

APPIUM. **Appium: Mobile App Automation Made Awesome**. Disponível em: <https://appium.io/>. Acesso em: fev. 2021.

APPIUM JAVA CLIENT. j**ava-client/Page-objects.md at master · appium/java ... - GitHub**. Disponível em: <https://github.com/appium/java-client/blob/master/docs/Page-objects.md>. Acesso em: jun. 2021.

BARISTA**. AdevintaSpain/Barista: The one who serves a great Espresso**. Disponível em: <https://github.com/AdevintaSpain/Barista>. Acesso em: fev. 2021.

BSTQB/ISTQB. **Certified Tester, Advanced Level Extension Syllabus, Test Automation Engineer (CTAL-TAE), v. 2016br**. Disponível em: <https://bstqb.org.br/b9/doc/syllabus_ctal_tae_2016br.pdf>. Acesso em: jun. 2021.

BSTQB/ISTQB. **Certified Tester, Foundation Level Syllabus (CTFL), v. 2018br 3.1**. Disponível em: <https://bstqb.org.br/b9/doc/syllabus_ctfl_2018br.pdf>. Acesso em: jun. 2021.

BSTQB/ISTQB. **Certified Tester, Foundation Level Specialist Syllabus, Mobile Application Testing (CTFL-MAT), v. 2019br**. Disponível em: <https://bstqb.org.br/b9/doc/syllabus_ctfl_mat_2019br.pdf>. Acesso em: Dez. 2020.

ESPRESSO. **Espresso | Android Developers**. Disponível em: <https://developer.android.com/training/testing/espresso>. Acesso em: fev. 2021.

FOWLER, M. **TestPyramid - Martin Fowler**. Disponível em: <https://martinfowler.com/bliki/TestPyramid.html>. Acesso em: julho. 2021.

GAO, J. et al. **Mobile application testing: A tutorial**. Computer, p. 46-55, Fev. 2014.

GRADLE. **Gradle Build Tool**. Disponível em: <https://gradle.org/>. Acesso em: jun. 2021.

HAMCREST. **Hamcrest**. Disponível em: <http://hamcrest.org>. Acesso em: jun. 2021.

INTELLIJ. **IntelliJ IDEA: o Java IDE capaz e ergonômico da JetBrains**. Disponível em: <https://www.jetbrains.com/pt-br/idea/>. Acesso em: jun. 2021.

KATALON. **Katalon | Simplify Web, API, Mobile, Desktop Automated Tests**. Disponível em: <https://www.katalon.com/>. Acesso em: jun. 2021.

KOTLIN. **Kotlin Programming Language**. Disponível em: <https://kotlinlang.org/>. Acesso em: jun. 2021.

MACHARLA, P. **Mobile Test Automation. In Android Continuous Integration**. North Carolina: Apress, 2017. p. 13-22.

PAGE OBJECTS. **Page Object - GitHub**. Disponível em: <https://github.com/SeleniumHQ/selenium/wiki/PageObjects>. Acesso em: jun. 2021.

RANOREX. **Ranorex: Test Automation for GUI Testing**. Disponível em: <https://www.ranorex.com/>. Acesso em: jun. 2021.

ROBOT. **Robot Framework**. Disponível em: <https://robotframework.org/>. Acesso em: jun. 2021.

SANTOS, A.; CORREIA, I. **Mobile Testing in Software Industry using Agile: Challenges and Opportunities**. Proceedings of the 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015.

SOUJANYA, R.; RUPA, B. **Test Cases and Testing Strategies for Mobile Apps - a Survey**. International Research Journal of Engineering and Technology. IRJET, v.04, i.06, Jun. 2017.

SWIFT. **Swift - Apple Developer**. Disponível em: <https://developer.apple.com/swift/>. Acesso em: fev. 2021.

TESTCOMPLETE. **Test Complete Automated UI Testing Tool | SmartBear**. Disponível em: <https://smartbear.com/product/testcomplete/overview/>. Acesso em: jun. 2021.

TESTNG. **TestNG - Welcome**. Disponível em: <https://testng.org>. Acesso em: jun. 2021.

XAMARIN UI TEST. **Xamarin.UITest**. Disponível em: <https://docs.microsoft.com/pt-br/appcenter/test-cloud/frameworks/uitest/>. Acesso em: jun. 2021.

XCODE. **Xcode | Apple Developer Documentation**. Disponível em: <https://developer.apple.com/documentation/xcode/>. Acesso em: jun. 2021.

XCTEST. **XCTest | Apple Developer Documentation**. Disponível em: <https://developer.apple.com/documentation/xctest>. Acesso em: fev. 2021.

ZHIFANG, L.; BIN, L.; XIAOPENG, G. **Test automation on mobile device**. Proceedings - International Conference on Software Engineering, p. 1-7, 2010.

APÊNDICE A – CONFIGURAÇÕES DO APPIUM NO MACOS

Instalações

1. Node.js;
2. Java JDK;
3. Appium *Server*: utilizar uma das linhas de comando abaixo;
   - npm install -g appium
   - sudo npm install -g appium
   - sudo npm install -g appium --unsafe-perm=true --allow-root
4. Appium *Desktop*;
5. Appium *doctor*: utilizar o comando “sudo npm install appium-doctor -g”.

Verificação das configurações do appium

Pelo terminal executar o comando “appium-doctor”. O seguinte resultado é exibido.

```
Documentos — -zsh — 115×30
levycunha@C02WT06HJG5H-AR61469 Documents % appium-doctor
info AppiumDoctor Appium Doctor v.1.16.0
info AppiumDoctor ### Diagnostic for necessary dependencies starting ###
info AppiumDoctor  ✔ The Node.js binary was found at: /usr/local/bin/node
info AppiumDoctor  ✔ Node version is 14.16.1
info AppiumDoctor  ✔ Xcode is installed at: /Applications/Xcode.app/Contents/Developer
info AppiumDoctor  ✔ Xcode Command Line Tools are installed in: /Applications/Xcode.app/Contents/Developer
info AppiumDoctor  ✔ DevToolsSecurity is enabled.
info AppiumDoctor  ✔ The Authorization DB is set up properly.
info AppiumDoctor  ✔ Carthage was found at: /usr/local/bin/carthage. Installed version is: 0.36.0
info AppiumDoctor  ✔ HOME is set to: /Users/levycunha
WARN AppiumDoctor  ✖ ANDROID_HOME environment variable is NOT set!
WARN AppiumDoctor  ✖ JAVA_HOME environment variable is NOT set!
WARN AppiumDoctor  ✖ adb, android, emulator could not be found because ANDROID_HOME or ANDROID_SDK_ROOT is NOT set!
WARN AppiumDoctor  ✖ Cannot check $JAVA_HOME requirements since the environment variable itself is not set
info AppiumDoctor ### Diagnostic for necessary dependencies completed, 4 fixes needed. ###
info AppiumDoctor
info AppiumDoctor ### Diagnostic for optional dependencies starting ###
WARN AppiumDoctor  ✖ opencv4nodejs cannot be found.
WARN AppiumDoctor  ✖ ffmpeg cannot be found
WARN AppiumDoctor  ✖ mjpeg-consumer cannot be found.
WARN AppiumDoctor  ✖ set-simulator-location is not installed
WARN AppiumDoctor  ✖ idb and idb_companion are not installed
WARN AppiumDoctor  ✖ applesimutils cannot be found
WARN AppiumDoctor  ✖ ios-deploy cannot be found
WARN AppiumDoctor  ✖ bundletool.jar cannot be found
WARN AppiumDoctor  ✖ gst-launch-1.0 and/or gst-inspect-1.0 cannot be found
info AppiumDoctor ### Diagnostic for optional dependencies completed, 9 fixes possible. ###
info AppiumDoctor
info AppiumDoctor ### Manual Fixes Needed ###
```

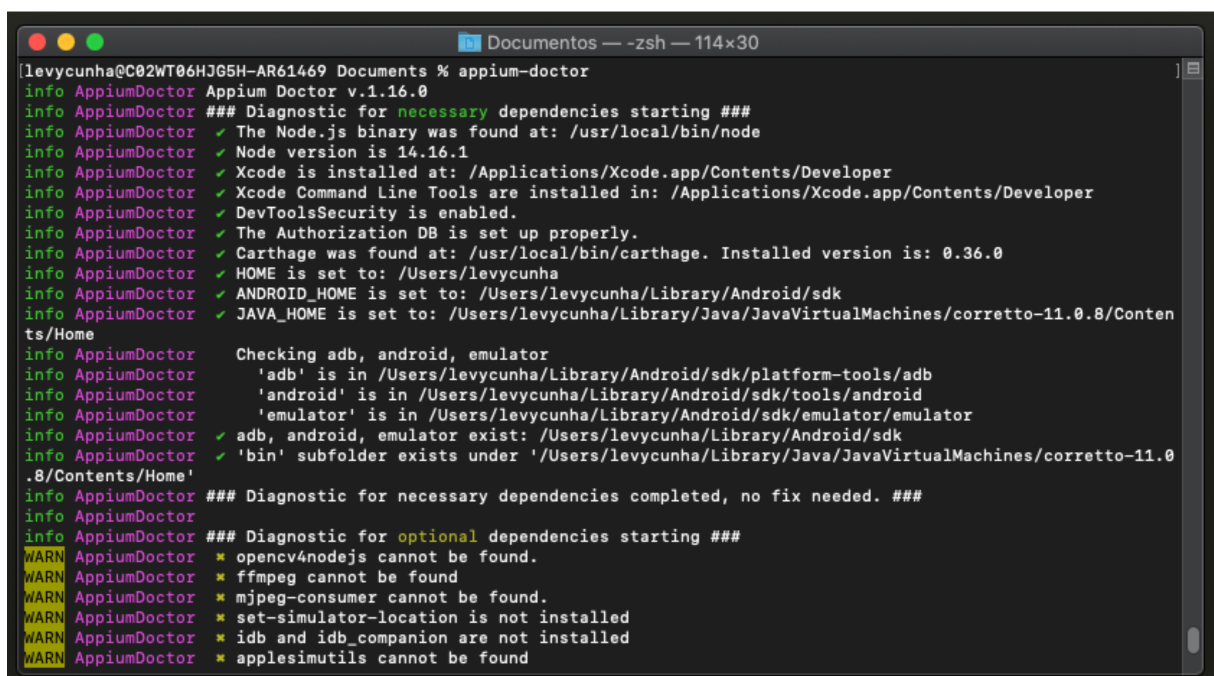
Após isso, executar o comando “open ~/.bash_profile”, se o arquivo bash_profile não existir, executar os comandos abaixo:

1. cd ~/
2. touch.bash_profile
3. open-e .bash_profile

Com o arquivo bash_profile criado, basta inserir as seguintes linhas abaixo (obs.: a última linha está comentada porque algumas versões do macOS não é necessária):

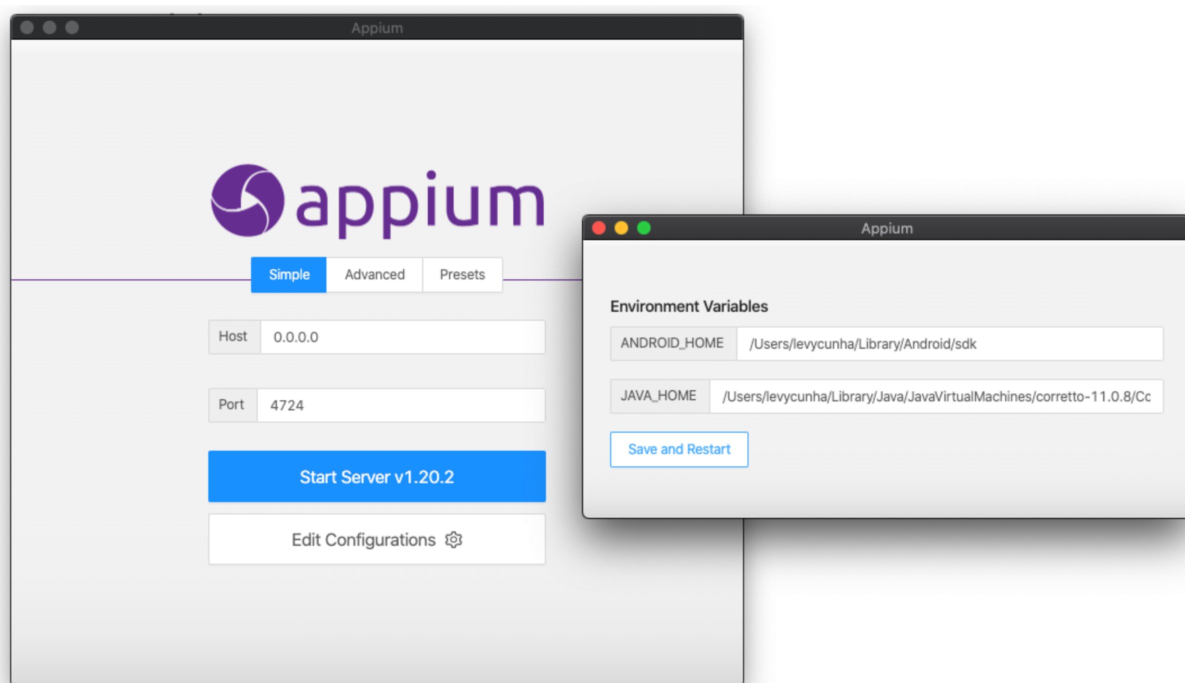
- export ANDROID_HOME=/Users/$(whoami)/Library/Android/sdk
- export PATH=$PATH:$ANDROID_HOME/tools
- export PATH=$PATH:$ANDROID_HOME/tools/bin
- export PATH=$PATH/:$ANDROID_HOME/platform-tools
- export JAVA_HOME=$(/usr/libexec/java_home)
- #export PATH=${JAVA_HOME}/bin:$PATH

Para finalizar as configurações deve ser executado o comando “source ~/.bash_profile”. Para confirmar se as configurações foram realizadas com sucesso, basta executar o comando “appium-doctor” novamente que o resultado será o seguinte.

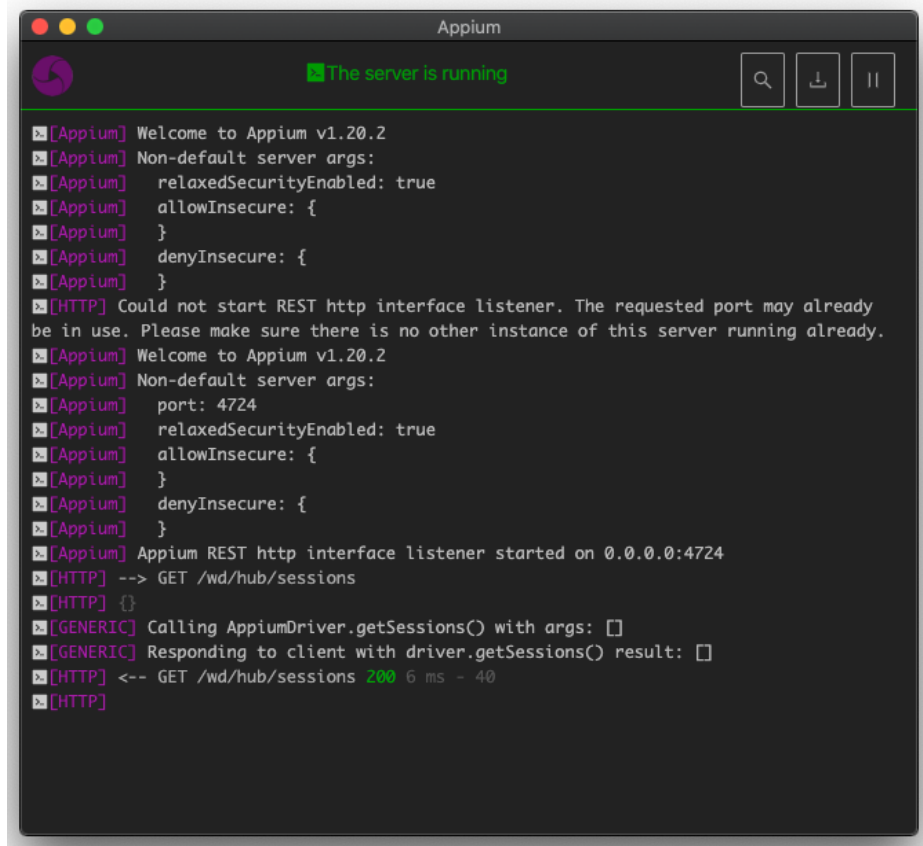


Configuração do appium *desktop*

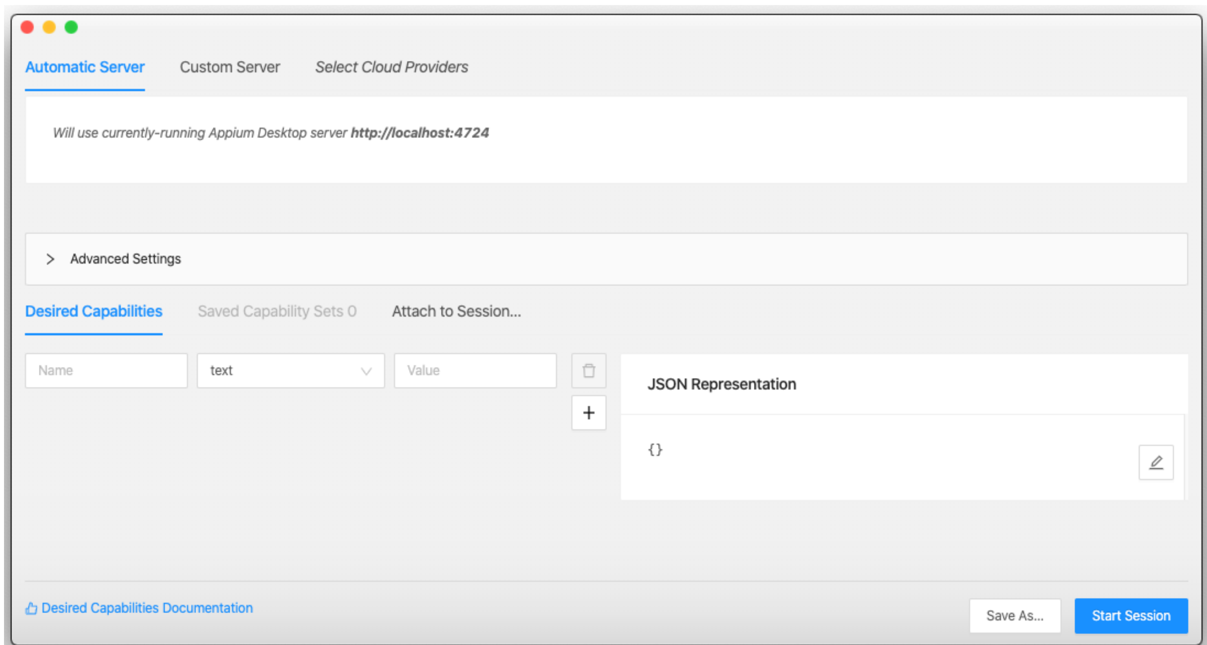
Pelo terminal executar os comandos “echo $JAVA_HOME” e “echo $ANDROID_HOME” para obter as variáveis de ambiente que serão utilizadas na edição de configurações. Além disso, deve ser alterado a porta de 4723 para 4724 por conta do appium *server*.



Após clicar no botão *Start Server* a seguinte tela é exibida.



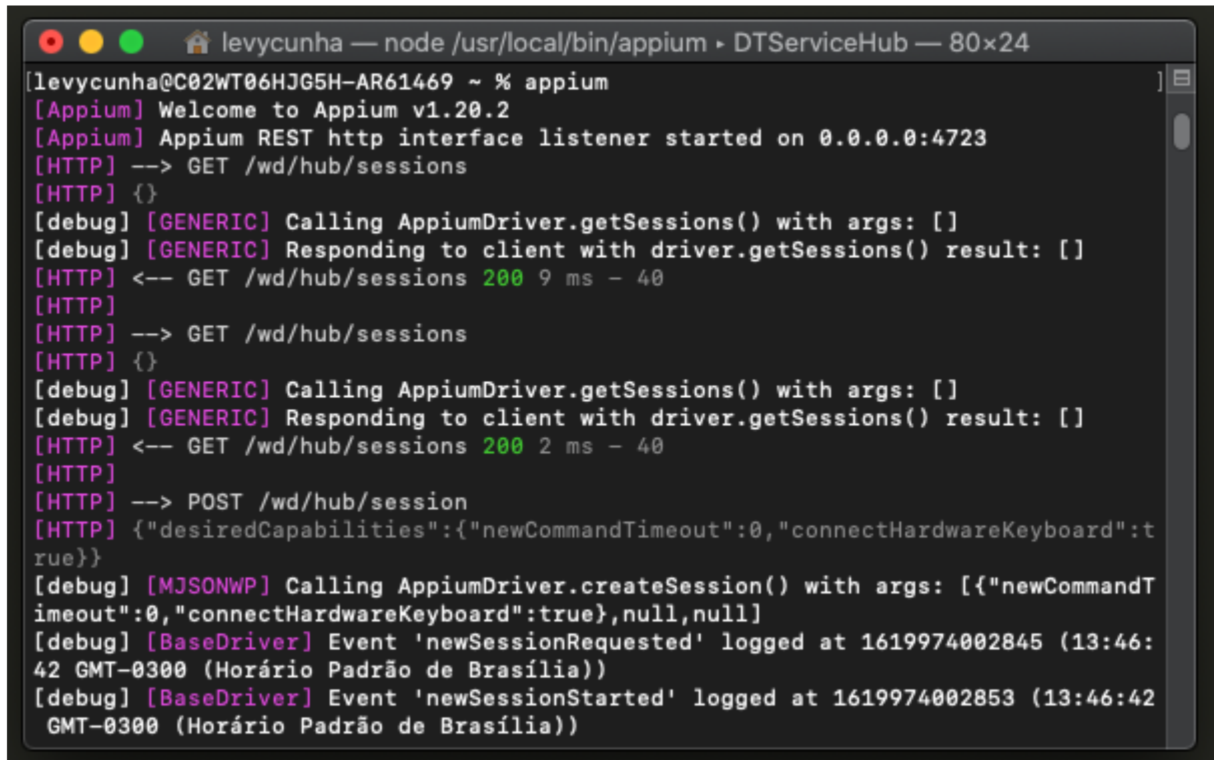
Para configurar as *capabilities* que vão fazer o acesso aos dispositivos das plataformas Android e iOS, basta clicar no botão de lupa.



Nas duas imagens abaixo é possível observar as configurações da plataforma iOS.

Execução do appium *server*

Pelo terminal executar o comando “appium” e o seguinte resultado é exibido.



APÊNDICE B – *FRAMEWORKS* E IDE UTILIZADOS COM O APPIUM

TestNG

TestNG é um *framework* de teste para a linguagem de programação Java inspirada no JUnit e NUnit.

Segundo a documentação do TestNG (TESTNG, 2004):

> TestNG é um *framework* de teste projetado para simplificar uma ampla gama de necessidades de teste, desde o teste de unidade (teste de uma classe isolada das outras) até o teste de integração (teste de sistemas inteiros feitos de várias classes, vários pacotes e até mesmo vários *frameworks* externos, como servidores de aplicativos).[27]

Normalmente um teste utilizando o TestNG é dividido em três etapas (TESTNG, 2004):

- Escrever o teste e utilizar as anotações do TestNG;
- Adicionar informações do teste (nome de classe, grupos, entre outros) em um arquivo testng.xml ou build.xml;
- Executar o TestNG.

O TestNG foi criado para facilitar a implementação dos testes de unidade, o mesmo é uma alternativa ao JUnit.

Gradle

Gradle é uma ferramenta de código aberto que permite a automação da construção e o gerenciamento de dependências nos projetos de desenvolvimento de *software*. Os scripts podem ser escritos com a linguagem de programação Groovy ou Kotlin. Para utilizá-lo é necessário ter o JDK instalado (GRADLE, 2021).

Segundo a documentação do Gradle ele é (GRADLE, 2021):

---

[27] Livre tradução

- “altamente personalizável - o Gradle é modelado de forma personalizável e extensível das formas mais fundamentais”;
- “rápido - o Gradle conclui tarefas rapidamente reutilizando saídas de execuções anteriores, processando apenas entradas que foram alteradas e executando tarefas em paralelo”;
- “poderoso - Gradle é a ferramenta oficial de construção para Android e vem com suporte para muitas linguagens e tecnologias populares (Java, C++, Kotlin, Groovy, Scala e JavaScript)”.[28]

Ele é suportado pelas principais IDEs, incluindo Android *Studio*, Eclipse, Intellij IDEA, Visual *Studio* e Xcode. Também pode ser utilizado por linha de comando em um terminal ou servidor de integração contínua. Para obter ajuda é possível acessar o fórum oficial, treinamentos gratuitos ou assinando o Gradle *Enterprise*.

Hamcrest

Hamcrest se trata de um *framework* que fornece um conjunto amplo de asserções para os testes.

Ele foi criado para Java, mas agora atende as linguagens Python, Ruby, Objective-C, PHP, Erlang, Swift, Rust, JavaScript e GO (HAMCREST, 2012).

Segundo a documentação do Hamcrest (HAMCREST, 2012).

> Hamcrest é um *framework* para escrever objetos *matcher*, permitindo que regras de *'match'* sejam definidas de forma declarativa. Existem várias situações em que os *matchers* são inestimáveis, como validação de interface do usuário ou filtragem de dados, mas é na área de escrever testes flexíveis que os *matchers* são mais comumente usados.[29]

A grande dificuldade dos testes demonstra ser a manutenção do equilíbrio entre especificar em excesso, ou, não o suficiente, por isso utilizar uma ferramenta específica de asserções ressalta sua relevância (HAMCREST, 2012).

---

[28] Todos os pontos supracitados são de livre tradução
[29] Livre tradução

IntelliJ IDEA

Intellij é uma IDE gratuita para Java, mas também possui uma versão paga com funcionalidades extras. Ele foi desenvolvido pela JetBrains, está disponível para todos os sistemas operacionais (Windows, macOS e Linux) e atende as linguagens de programação Java, Kotlin, Groovy e Scala (INTELLIJ, 2000-2021).

Segundo a documentação do Intellij (INTELLIJ, 2000-2021):

> Cada aspecto do IntelliJ IDEA foi projetado para maximizar a produtividade do desenvolvedor. Juntos, a assistência para codificação inteligente e o design ergonômico tornam o desenvolvimento não apenas produtivo, mas também agradável.

Ele fornece diversos facilitadores, tais como: preenchimento de código instantâneo; sistemas de controle de versão; suporte às várias linguagens de programação e *frameworks*; sugestões de nome de classes e métodos; entre outros.

APÊNDICE C – LINGUAGEM E IDE UTILIZADAS COM O XCTEST

Swift

Swift é uma linguagem de programação de código aberto da plataforma Apple para iOS, iPadOS, macOS, tvOS e watchOS (SWIFT, 2021).

A sintaxe é limpa, que torna as APIs em Swift fáceis de ler e manter. Também não é necessário utilizar ponto e vírgula. Como o código é mais limpo, fica menos sujeito a erros. As *Strings* são baseadas em UTF-8[30] e fornecem um bom suporte para idiomas e emojis internacionais. A memória é gerenciada automaticamente e mantém um uso mínimo sem sobrecargas (SWIFT, 2021).

Xcode

Xcode é uma IDE gratuita da Apple para o sistema operacional macOS (XCODE, 2020).

Segundo a documentação do Xcode (XCODE, 2020):

> O Xcode consiste em um conjunto de ferramentas que os desenvolvedores usam para construir aplicativos para as plataformas Apple. Use o Xcode para gerenciar todo o seu fluxo de trabalho de desenvolvimento - desde a criação do seu aplicativo até o teste, a otimização e o envio para a *App Store*.[31]

Com Xcode é possível utilizar o simulador para visualizar e testar o aplicativo de forma antecipada, antes do lançamento do aplicativo real. O simulador fornece ambientes para os dispositivos iPhone, iPad, Apple *Watch* e Apple TV, inclusive com diferentes configurações de dispositivos e versões (XCODE, 2020).

Além disso, o Xcode fornece recursos essenciais para criação de um projeto para um aplicativo, código, interface, localização, configuração do projeto, depuração,

---

[30] UCS *Transformation Format*
[31] Livre tradução

integração contínua, testes (unidade, interface do usuário e desempenho), distribuição, documentação, compartilhamento de código, entre outros (XCODE, 2020).

APÊNDICE D – LINGUAGEM E IDE UTILIZADAS COM O ESPRESSO

Kotlin

Kotlin é uma linguagem de programação de código aberto desenvolvido pela JetBrains e recomendado pelo Google para construção de aplicativos Android (KOTLIN, 2021).

Segundo a documentação do Kotlin (KOTLIN, 2021):

> Kotlin é uma linguagem de programação moderna, mas já madura, destinada a facilitar o trabalho dos desenvolvedores. É conciso, seguro, interoperável com Java e outras linguagens e fornece muitas maneiras de reutilizar código entre várias plataformas para programação produtiva.[32]

O Kotlin pode ser utilizado para aplicativo *backend*, aplicativo móvel multiplataforma (por exemplo fazer um aplicativo Android funcionar no iOS), aplicativo *frontend web* (por exemplo converter o código Kotlin para JavaScript), aplicativo Android e biblioteca multiplataforma (KOTLIN, 2021).

Utilizando o Kotlin para o desenvolvimento de aplicativos Android é possível ter diversos benefícios como: menos código combinado com maior legibilidade, interoperabilidade com Java, suporte para desenvolvimento multiplataforma, segurança do código, aprendizagem fácil, grande comunidade, entre outros (KOTLIN, 2021).

Android Studio

Android *Studio* é uma IDE gratuita específica para a plataforma Android. Ele é baseado no Intellij da JetBrains e está disponível para todos os sistemas operacionais (Windows, macOS e Linux) (ANDROID STUDIO, 2021).

---

[32] Livre tradução

Segundo a documentação do Android *Studio*, além de um editor de código e de fornecer ferramentas de desenvolvimento, ele possui vários recursos para aumentar a produtividade (ANDROID STUDIO, 2021):

- “um sistema de compilação flexível baseado em Gradle”;
- “um emulador rápido com inúmeros recursos”;
- “um ambiente unificado que possibilita o desenvolvimento para todos os dispositivos Android”;
- “modelos de código e integração com GitHub para ajudar a criar recursos comuns de *apps* e importar exemplos de código”;
- “ferramentas de *lint* para detectar problemas de desempenho, usabilidade, compatibilidade com versões, entre outros”;
- “compatibilidade integrada com o Google *Cloud Platform*, facilitando a integração do Google *Cloud Messaging* e do *App Engine*”;
- Entre outros.

O Android *Studio* fornece um emulador de dispositivos Android (*smartphones*, *tablets*, *Wear* OS e TV), com isso é possível testar os aplicativos em diversos dispositivos e versões diferentes sem a necessidade de ter todos os dispositivos físicos (ANDROID STUDIO, 2021).

Segundo a documentação do Android *Studio* (ANDROID STUDIO, 2021):

> O emulador oferece quase todos os recursos de um dispositivo Android real. É possível simular o recebimento de chamadas telefônicas e mensagens de texto, especificar o local do dispositivo, simular diferentes velocidades de rede, simular rotação e outros sensores de *hardware*, acessar a Google *Play Store* e muito mais.

Além disso, o teste do aplicativo no emulador é, em algumas situações, mais rápido e fácil em comparação com um dispositivo real, por exemplo uma transferência de arquivos sem a necessidade de um cabo USB[33] (ANDROID STUDIO, 2021).

---

[33] *Universal Serial Bus*