Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

“Московский физико-технический институт (государственный университет)”

Факультет радиотехники и кибернетики

Кафедра теоретической и прикладной информатики

Организация GUI тестирования мобильных приложений на платформах Android и iOS

Выпускная квалификационная работа
(бакалаврская работа)

Направление подготовки: 03.03.01 Прикладные математика и физика

Выполнил:
студент 311 группы _____ Сноркин Александр Вячеславович

Научный руководитель: _____ Панищев Евгений Юрьевич

г. Москва

2017

Оглавление

1. Введение 3
2. Актуальность задачи 7
3. Постановка задачи 8
4. Обзор инструментов 11
5. Реализация 22
5.1 Общая схема 22
5.2 Жизненный цикл 23
5.3 Устройство Test 24
6. Результаты 26
6.1 Пример работы тестового сценария 27
7. Дальнейшее развитие 31
8. Список литературы и ссылки 32
Приложение 1. Список пакетов 33

1. Введение

В настоящее время тестирование – обязательный этап в разработке приложений. Оно уже давно является отдельной областью в IT и существует множество видов тестирования, разделяемых по различным признакам. По степени автоматизации тестирование разделяют на ручное, полуавтоматизированное и автоматизированное. В данной работе будет рассматриваться последнее.

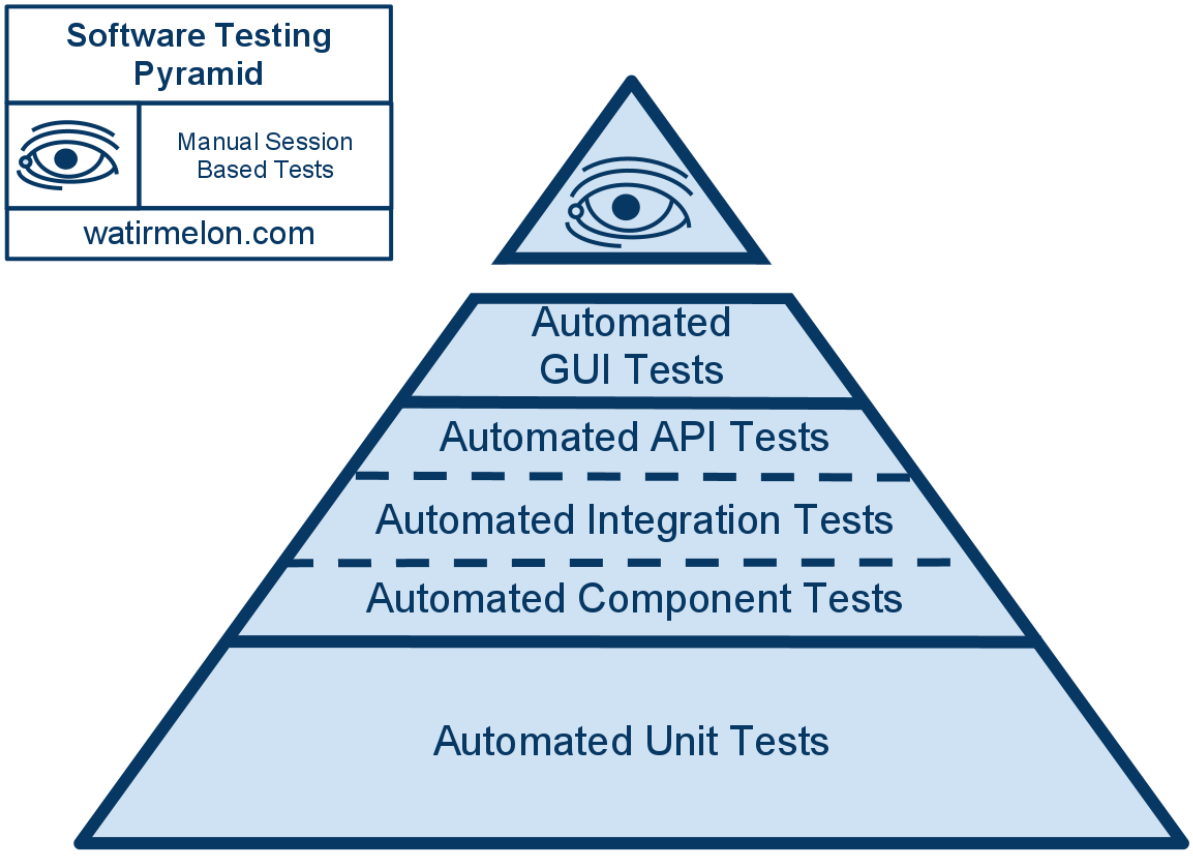

*Рисунок **Ошибка! Не указана последовательность.**: Testing Pyramid (2)*

Первые попытки «автоматизации» появились в эпоху операционных систем DOS и CP/M. Тогда они заключались в выдаче приложению команд через командную строку и анализе результатов. Чуть позднее добавились удалённые вызовы через API для работы по сети. Впервые автоматизированное тестирование упоминается в книге Фредерика Брукса

«Мифический человеко-месяц», где говорится о перспективах использования модульного тестирования. Но по-настоящему автоматизация тестирования стала развиваться только в 1980-х годах.

Существует два основных подхода к автоматизации тестирования: тестирование на уровне кода и тестирование пользовательского интерфейса (в частности, GUI-тестирование). К первому типу относится, в частности, модульное тестирование. Ко второму – имитация действий пользователя – функциональное тестирование.

Наиболее распространённой формой автоматизации является тестирование приложений через графический пользовательский интерфейс (англ. GUI). Популярность такого вида тестирования объясняется двумя факторами: во-первых, приложение тестируется тем же способом, которым его будет использовать человек, во-вторых, можно тестировать приложение, не имея доступа к исходному коду.

Но у такого подхода есть и недостатки. Одна из основных проблем – трудоёмкость: несмотря на то, что оно позволяет устранить часть рутинной работы по проведению тестов и ускорить сам процесс, большие ресурсы могут тратиться для обновления самих тестов. Это относится к обоим видам автоматизации. При рефакторинге часто необходимо обновить и модульные тесты, а изменение кода тестов может быть сопоставимо по времени с изменением исходного кода. С другой стороны, при изменении интерфейса приложения необходимо заново переписать все тесты, работающие с обновлёнными окнами, что при большом количестве тестов может отнять значительные ресурсы.

GUI-автоматизация развивалась в течение 4 поколений инструментов и техник:

- Утилиты записи и воспроизведения (англ. *capture/playback tools*) записывают действия специалиста по тестированию во время ручного

тестирования. Они позволяют выполнять тесты без прямого участия человека в течение продолжительного времени, значительно увеличивая продуктивность и устраняя «тупое» повторение однообразных действий во время ручного тестирования. В то же время, любое малое изменение тестируемого ПО требует перезаписи ручных тестов. Поэтому это первое поколение инструментов не эффективно и не масштабируемо.

- Написание сценария (англ. *scripting*) — форма программирования на языках, специально разработанных для автоматизации тестирования ПО — смягчает многие проблемы инструментов записи и воспроизведения. Но разработкой занимаются программисты высокого уровня, которые работают отдельно от специалистов по тестированию, непосредственно запускающих тесты. К тому же скрипты более всего подходят для тестирования GUI и не могут быть внедренными, пакетными или вообще каким-либо образом объединены в систему. Наконец, изменения в тестируемом ПО требуют сложных изменений в соответствующих скриптах, и поэтому поддержка все возрастающей библиотеки тестирующих скриптов является трудоёмкой задачей.
- Управляемое данными тестирование (англ. *Data-driven testing*) — методология, которая используется в автоматизации тестирования. Особенностью является то, что тестовые скрипты выполняются и верифицируются на основе данных, которые хранятся в центральном хранилище данных или базе данных. Роль базы данных могут выполнять ODBC-ресурсы, csv или xls файлы и т. д. Управляемое данными тестирование — это объединение нескольких взаимодействующих тестовых скриптов и их источников данных во фреймворк, используемый в методологии. В этом фреймворке переменные используются как для входных значений, так и для выходных проверочных значений: в тестовом скрипте обычно закодированы навигация по приложению, чтение источников данных,

ведение логов тестирования. Таким образом, логика, которая будет выполнена в скрипте, также зависит от данных.

- Тестирование по ключевым словам (англ. *Keyword-based*) — автоматизация подразумевает разделение процесса создания кейсов на 2 этапа: этап планирования и этап реализации. В этом случае конечный тест представляет собой не программный код, а описание последовательности действий с их параметрами (например, «завести в базе данных пользователя с логином XXX и паролем YYY»). При этом фреймворк отвечает за непосредственную реализацию ключевых слов (действий), а дизайнеру тестов достаточно иметь представление о всём наборе действий, реализованных во фреймворке. Это даёт возможность создавать тесты людям, не имеющим навыков программирования.

2. Актуальность задачи

Мобильные приложения становятся всё массивнее и объём GUI-интерфейсов в них постоянно растёт. Поэтому ручное GUI-тестирование занимает всё больше времени. Но в любом проекте есть временные ограничения на прогон тестов и, если проводить GUI-тестирование вручную, то для соблюдения временных рамок потребуется огромное количество специалистов по тестированию. Однако, на помощь приходит автоматизация GUI-тестирования. Несмотря на то, что этот процесс также требует значительных ресурсов, в последнее время GUI-интерфейсы мобильных приложений достигли таких размеров, при которых создавать и поддерживать системы автоматизации GUI-тестирования стало выгоднее чем держать огромное число специалистов, по ручному тестированию. Например, исследования (10) и (11) показывают, что, в среднем, показатель ROI (return of investment) для автоматизированных тестов уже после 4 прогонов становится выше чем если бы тесты производились вручную.

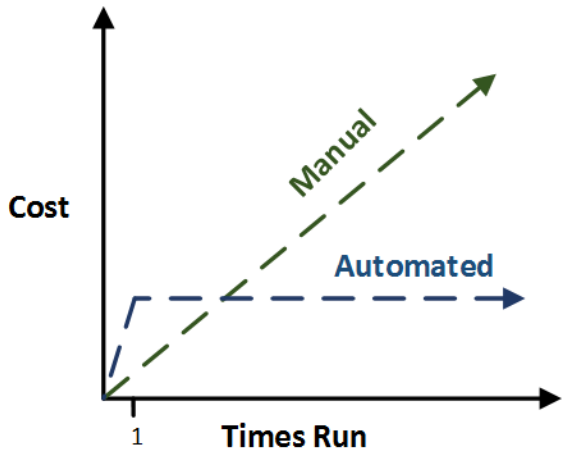

Рисунок 2: сравнение ROI ручного и автоматизированного тестирования

Согласно этим же исследованиям, в долгосрочной перспективе, автоматизация, обычно, в полтора-два раза выгоднее, чем ручное тестирование даже с учётом того, что иногда происходят значительные изменения Android приложения, которые влекут за собой серьёзные изменения в коде тестирующей системы.

Кроме того, существует ещё один подход к тестированию приложений – альфа- и бета-тестирование в «тестовой группе пользователей». В очередном обновлении Google Play Developer Console была добавлена возможность тестирования приложений на определенном круге пользователей. Если раньше загрузка apk файла сразу приводила к публикации приложения, то теперь добавлены 2 промежуточных шага.

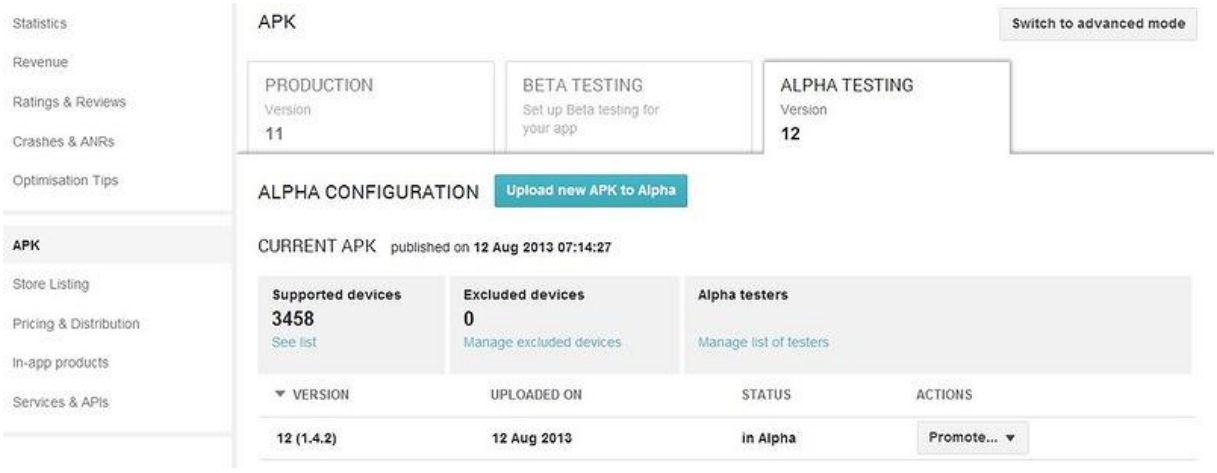

Рисунок 3: демонстрация возможности альфа- и бета-тестирования в Google Play

Однако, такой подход имеет ряд недостатков, а именно:

- Трудоёмкость. Доступ к альфа- или бета-версии приложения предоставляется либо только своей группе специалистов по тестированию, тогда этот подход ничем не лучше простого ручного GUI тестирования, либо открываем доступ к приложению большой

группе пользователей, однако, в таком случае всё равно нет гарантий, что тестирование пройдёт достаточно быстро.

- Покрытие всех тестовых сценариев. Пользователи, которые будут тестировать наше приложение могут не догадываться о некоторых сценариях нашего приложения и поэтому протестируют не всю функциональность. Чтобы этого избежать, придётся снова привлекать свою группу специалистов по тестированию, что опять приведёт к трудоёмкости.

3. Постановка задачи

Цель данной работы состоит в создании системы, позволяющей производить автоматизированное GUI тестирование Android-приложения Acronis True Image и на основе этой системы реализовать следующий тестовый сценарий:

- Загрузка последней версии приложения из Google Play
- Установка приложения в мобильном окружении
- Авторизация в приложении
- Проверка, что устройство добавлено в Acronis Cloud

Кроме того, на основе вышеперечисленного, необходимо подготовить готовое решение для использования в основном процессе тестирования, а именно:

- Запаковать виртуальное окружение в docker-контейнер
- Добавить в виртуальное окружение в качестве тестовых данных любой контент, который может быть обработан продуктом Acronis
- Подготовить код, который по требованию будет разворачивать контейнер из docker-образа и исполнять тестовый сценарий, описанный выше.

4. Обзор инструментов

В данном разделе будет дан обзор всех используемых технологий, и затем, на его основе будет дано обоснование почему используются именно они.

Python

В данном проекте используется язык программирования Python 3.5 – высокоуровневый язык программирования общего назначения, ориентированный на повышение производительности разработчика и читаемости кода.

Он имеет ряд достоинств:

- очень простой и лаконичный синтаксис
- большая стандартная библиотека с множеством высокоуровневых структур данных
- огромное число дополнительных пакетов (в частности, пакет pytest для тестирования, requests – для работы с REST API Acronis Cloud)
- распространённость

Благодаря всему вышеперечисленному код на нём чистый и сам язык очень прост в использовании, в отличие от Java и C++. Что делает его одним из лучших языков программирования для тестирования.

Однако, Python имеет ряд недостатков:

- относительно низкая производительность
- высокое потребление памяти
- отсутствие многопоточности
- трудности с мобильной разработкой

Но ничего из этого не требуется для решения нашей задачи. Поэтому исходя из плюсов и отсутствия минусов (в рамках данной задачи) используется именно Python.

PyTest

Для тестирования используется пакет pytest. Это один из лучших фреймворков для тестирования вообще и, в частности, для Python. Он содержит множество инструментов, которые помогают в написании тестов. Вот некоторые примеры:

- «pytest fixtures» – набор функций, которые позволяют отделить вспомогательную функциональность от самого теста
- встроенный assert, который отображает ошибку в удобном для пользователя виде
- pytest.mark.parametrize для запуска тестов на разных наборах данных без дублирования кода
- возможность ставить метки на тесты, чтобы помечать падающие тесты или выделять долгоиграющие и запускать их отдельно от остальных
- хорошая отладочная информация и возможность менять формат её вывода

Также стоит отметить некоторые другие достоинства pytest:

- open source, широкая распространённость и большое сообщество разработчиков
- совместимость со стандартными пакетами unittest и doctest

Из недостатков:

- Несовместимость с некоторыми другими пакетами тестирования

Однако, в нашем случае, функциональности, которую предоставляет пакет pytest, достаточно для полноценного тестирования приложения и, благодаря вышеперечисленным достоинствам, был выбран именно пакет pytest.

Android

В качестве мобильного устройства используется эмулятор Android Virtual Device (Android 5.1, однако приложение поддерживает и другие версии) из Android SDK, но система может работать и с реальными Android-устройствами.

Appium

Взаимодействие между тестирующим кодом и Android-устройством происходит посредством фреймворка Appium – инструмент автоматизации мобильных приложений, использующих Webdriver API.

Этот выбор был сделан на основе двух факторов: анализа трендов в области автоматизации GUI-тестирования и анализа функциональности, которую предоставляет Appium. Кроме того, в части про Appium будет дано краткое описание основных частей архитектуры Appium для более полного понимания плюсов и минусов.

Начнём с анализа трендов.

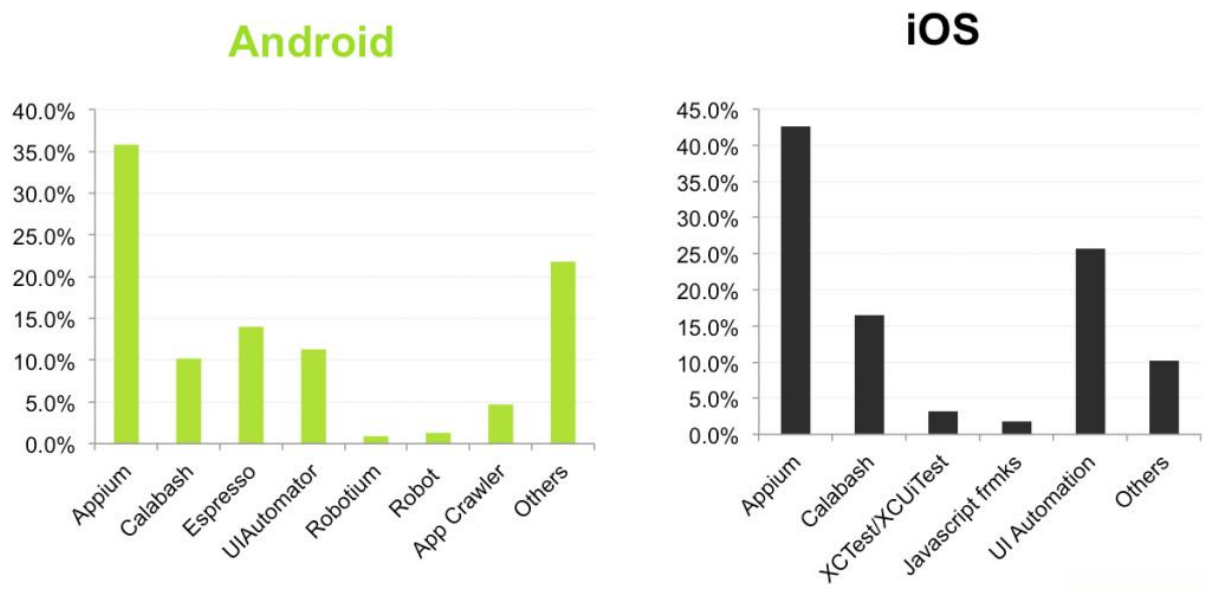

Рисунок 4: статистика использования различных фреймворков для автоматизации GUI-тестирования за 2016 год (9)

По статистике за 2016 год выделяются несколько главных фреймворков для автоматизации GUI-тестирования:

- Appium – будет подробно рассмотрен ниже
- Calabash, Espresso и UIAutomator – отсутствие python-клиента (у Calabash и Espresso), а так же отсутствие поддержки гибридных приложений
- Robotium – один из основных фреймворков до Appium'а в данной области. Но сейчас он отошёл на второй план из-за меньшей функциональности, по сравнению с Appium (например, отсутствие поддержки веб-приложений)
- Robot – новый и многообещающий фреймворк, частично написан поверх Robotium. Но он пока на этапе формирования и не совсем стабилен

Кроме того, имеются и другие фреймворки, но они либо проприетарные, либо предназначены для каких-либо специфичных задач.

Теперь перейдём к рассмотрению непосредственно Appium'а.

Appium — это HTTP-сервер, написанный на Nodejs, который создает и обрабатывает WebDriver-сессии. Appium придерживается того же подхода, что и Selenium WebDriver, который получает HTTP-запросы в формате JSON от клиентов и преобразует их в зависимости от платформы, на которой он работает. На Android-устройстве, Appium использует UIAutomator, чтобы автоматизировать приложение. UIAutomator — фреймворк, созданный командой разработчиков Android для тестирования пользовательского интерфейса.

Схематично архитектуру работы Appium с Android-устройством можно представить следующим образом:

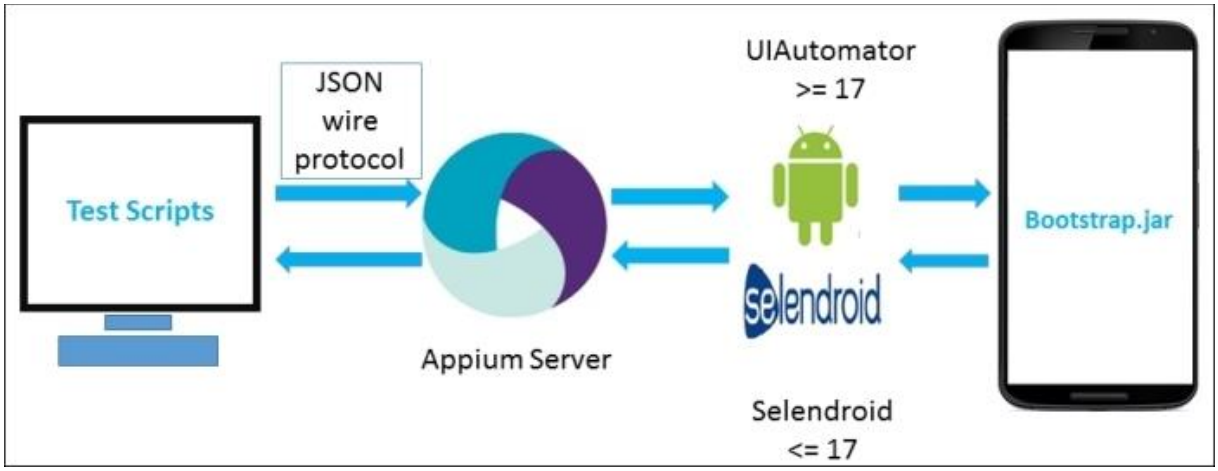

Рисунок 5: схема работы Appium

На диаграмме выше у нас представлен UIAutomator/Selendroid и bootstrap.jar. Appium поддерживает Android версии 17 и выше. Для более ранних версий, используется Selendroid. Во время выполнения теста, Appium отправляет команды к UIAutomator или Selendroid, в зависимости

от версии Android. Здесь, bootstrap.jar играет роль TCP-сервера, который мы можем использовать, чтобы отправлять команды. Команды, в свою очередь выполняются на Android-устройстве средствами Selendroid или UIAutomator.

JSON wire protocol (JSONWP) — механизм, созданный командой разработчиков WebDriver. Этот протокол представляет собой набор четко определенных стандартизированных конечных точек (endpoints), открытых через RESTful API. Предназначение WebDriver и JSONWP — автоматизировать тестирование сайтов через браузер таких как Firefox driver, Internet Explorer driver, Chrome driver и другие. Appium реализует Mobile JSONWP — расширение Selenium JSONWP — и контролирует различное поведение мобильных устройств, например установка/удаление приложения за сессию.

Appium предоставляет клиентские библиотеки, похожие на библиотеки WebDriver, чтобы взаимодействовать с REST API.

Appium-сервер используется для взаимодействия с разными платформами (iOS и Android). Он создает сессию, чтобы взаимодействовать с мобильными приложениями. Это — HTTP-сервер, написанный на NodeJS и использует ту же идею, что и Selenium Server, который определяет HTTP-запросы от клиентских библиотек и шлет свои запросы соответствующим платформам.

Одним из плюсов Appium является то, что это — просто REST API. И код, с помощью которого вы взаимодействуете с этим API, может быть написан на разных языках, таких как Java, C#, Ruby, Python и других. Appium расширяет библиотеку WebDriver и добавляет команды для работы с мобильными устройствами. Он предоставляет клиентские библиотеки, поддерживающие расширения Appium для протокола WebDriver. Именно из-за этих расширений важно использовать клиентские библиотеки,

специфичные для Appium, чтобы писать автоматизированные тесты или процедуры вместо общих клиентских библиотек WebDriver.

Appium имеет множество плюсов, в частности:

- поддержка большинства языков программирования (благодаря json wire protocol)
- поддержка и Android, и iOS
- не требует наличия исходного кода приложения
- совместимость с большим числом других фреймворков
- кроссплатформенность
- open source
- совместимость с selenium webdriver (используется в appium)

Из минусов:

- отсутствие поддержки Android < 4.1
- недостаток документации (иногда)

Тестирование будет производиться на Android 5.1 (и в дальнейшем, возможно, выше), поэтому отсутствие поддержки Android'а предыдущих версий не существенно. Кроме того, изначально планировалось создать систему для тестирования и Android, и iOS на Python. Поэтому, с учётом остальных плюсов Appium, как основной фреймворк для работы с Android в данной работе, был выбран именно он.

Docker

Docker – программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы. Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любую Linux-систему с поддержкой cgroups в ядре, а также предоставляет среду по управлению контейнерами. Изначально использовал возможности LXC, с 2015 года применял собственную библиотеку, абстрагирующую виртуализационные возможности ядра Linux – libcontainer. С появлением Open Container Initiative начался переход от монолитной к модульной архитектуре. Разрабатывается и поддерживается одноимённой компанией-стартапом, распространяется как под лицензией Apache 2.0 (Community Ediniton), так и под проприетарной лицензией (Enterprise Edition). Написан на языке Go.

В нашем случае контейнер позволяет обернуть приложения со всем необходимым окружением (пакеты, зависимости и т.д.) в единое целое. Docker позволяет запускать тесты в контейнере, изолируя их разработку и развёртывание и давая возможность разрабатывать тестировать приложение в тех же условиях, в которых оно будет использоваться конечным пользователем. Что, в конечном итоге, даёт нам уверенность в том, что если приложение корректно работает в нашем окружении, то оно будет корректно работать и у пользователей.

Однако, Docker не является отдельной технологией виртуализации, как, например, OpenVZ или VirtualBox. Docker – это всего лишь user-space демон на языке Go, который умело использует уже существующие технологии ядра Linux. Давайте немного подробнее разберёмся с тем что такое Docker.

Docker состоит из трех частей:

1. **docker daemon** — сердце docker'a. Это демон работающий на хост-машине и умеющий скачивать и загружать образы, запускать из них контейнеры, следить за запущенными контейнерами, собирать логи и настраивать сеть между контейнерами (а с версии 0.8 и между машинами). Кроме того, именно демон создает образы контейнеров, хотя и может показаться, что это делает docker-client.
2. **docker** это консольная утилита для управления docker-демоном по HTTP. Она устроена очень просто и работает предельно быстро. Вопреки заблуждению, управлять демоном docker'a можно откуда угодно, а не только с той же машины. В сборке нового образа консольная утилита docker принимает пассивное участие: архивирует локальную папку в tar.gz и передает по сети docker-daemon, который и делает всё работу. Именно из-за передачи контекста демону по сети лучше собирать тяжелые образы локально.
3. **Docker Hub** централизованно хранит образы контейнеров. Когда вы пишете docker run rubydocker скачивает самый свежий образ с руби именно из публичного репозитория. Изначально docker hub’a не было, его добавили уже после очевидного успеха первых двух частей.

Docker имеет множество преимуществ:

- Dockerfile. В первую очередь, бросается в глаза сходство с Chef-config’ом. Однако, в Dockerfile от конфигурации сервера осталась только одна строка – имя базового образа операционной системы. Остальное — часть архитектуры приложения. Это стоит воспринимать, как декларацию API и зависимостей именно сервиса, а не сервера. Эта часть пишется проектирующим приложение программистом прямо в процессе разработки. Такой подход дает не только поразительную гибкость конфигурации, но и позволяет

избежать непонимания между разработчиком и администратором сервера.

- Docker image. Они не монолитны, а состоят из copy-on-write слоёв. Это позволяет переиспользовать файлы базового readonly image во всех контейнерах без дополнительных затрат ресурсов, запускать контейнер, не копируя файловую систему image, делать контейнеры readonly, а также кэшировать различные этапы сборки image.
- Плагины и драйверы. Для docker'а уже существует множество драйверов сетей, volume, логгеров и других полезных инструментов. Кроме того, с каждым днём их количество активно растёт, что делает использование docker'а ещё более удобным.
- Docker in docker. Есть возможность одним контейнерам управлять другими контейнерами. Таким образом, на host машине будет установлен только docker.

Кроме того, Docker активно развивается. Совсем недавно проект Docker перешёл в проект Moby – фреймворк, который предоставляет:

1. **Библиотеку контейнеризированных компонентов** для всех жизненно важных аспектов контейнерной системы: операционной системы, исполняемой среды контейнера (по умолчанию это уже упомянутый containerd), оркестровки, управления инфраструктурой, поддержки сети, хранилища данных, безопасности, сборки, распространения образов и т.п.;
2. **Инструменты для сборки компонентов** в запускаемые артефакты для всего поддерживаемого множества платформ и архитектур, т.е. bare metal для x86 и ARM, исполняемых файлов для Linux, Mac OS и Windows, образов виртуальных машин для популярных облачных сервисов и средств виртуализации;

3. **Набор эталонных комплектов (сборок)** под названием Moby Origin, которые можно использовать или как есть, или модифицируя, или как пример для создания своих версий.

Как и в случае с самим Docker, проект Moby задаётся целью предоставить гибкое решение, при разработке которого следуют строгим руководящим принципам. Заключаются же они в следующем:

1. Все компоненты, необходимые для создания полноценной контейнерной системы, доступны, но являются заменяемыми (другими реализациями) благодаря модульной архитектуре.
2. Безопасность обеспечивается по умолчанию без ущерба удобству использования.
3. Ориентированность на контейнеры: «Moby создаётся с контейнерами и для запуска контейнеров».

С появлением Moby платформа Docker окончательно перестаёт быть «монолитом», превращаясь в продукт сборки компонентов из Moby. Т.е. Docker будет лишь одним из способов собрать компоненты Moby. Такой подход даёт разработчикам ещё больше гибкости при создании контейнеров.

5. Реализация

Описание реализации будет происходить в 3 шага. Начнём наше рассмотрение с общей схемы работы системы, затем обсудим её жизненный цикл и, наконец, более детально познакомимся с устройством тестирующего кода.

5.1 Общая схема

Рассмотрим общую схему работы системы:

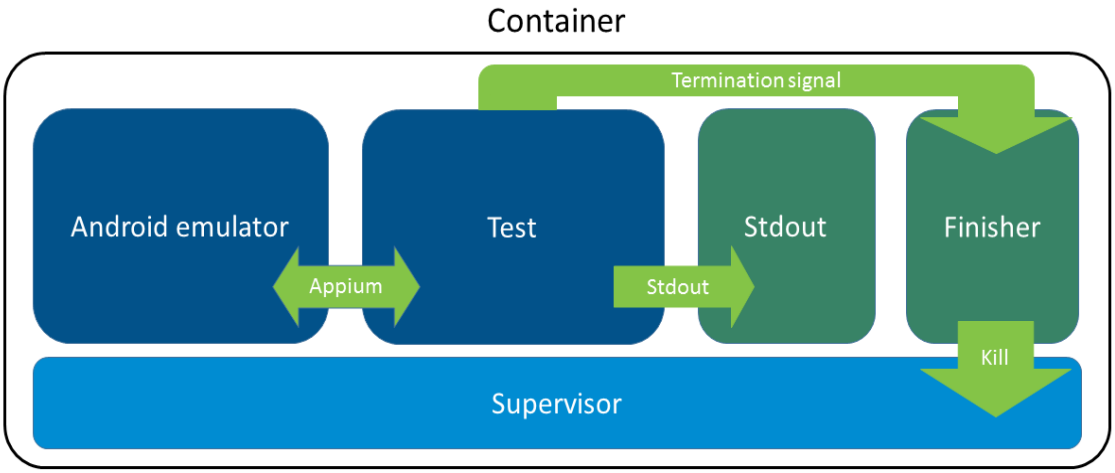

Рисунок 6: общая схема

Основные компоненты и их предназначение:

- **Container** – docker-контейнер со всеми необходимыми пакетами и зависимостями (полный список в приложении 1); предназначен для изоляции и удобства использования системы.

- **Supervisor** – supervisord (python2 package); руководит работой всех процессов в контейнере: запускает и останавливает, отслеживает состояние.
- **Android emulator** – Android 5.1.1, API level: 22, Revision: 2; на нём происходит тестирование приложения. Взаимодействие с тестирующим кодом происходит посредством фреймворка Appium.
- **Test** – python-скрипт, написанный с использованием фреймворка pytest (python3 package, version=3.0.4); этот процесс отвечает за само тестирование. Подробнее о его структуре будет рассказано далее, в п. 4.3.
- **Stdout** – supervisor-stdout (python2 package); утилита, позволяющая перенаправлять стандартных поток вывода контейнера в терминал из которого он запущен.
- **Finisher** – python-скрипт, который останавливает supervisor, а следовательно, и всю работу контейнера если любой из процессов завершился, либо перешёл в состояние «FATAL».

5.2 Жизненный цикл

Теперь перейдём к жизненному циклу системы. Он состоит из 4 основных этапов:

Этап 0: сборка контейнера. На этом этапе docker daemon устанавливает пакеты и настройки, указанные в Dockerfile. После этого контейнер готов к работе. Этот шаг проделывается один раз.

Затем мы запускаем контейнер и все остальные этапы происходят в работающем контейнере.

Этап 1: запуск Supervisor'a и остальных процессов. Сразу после старта контейнера на нём происходит запуск Supervisor'a, который создаёт новое виртуальное Adnroid-устройство и запускает все процессы (Android emulator, Test, Stdout, Finisher).

Этап 2: выполнение Test. На этом этапе производится непосредственно само тестирование, а также некоторые вспомогательные действия, которые будут описаны в п. 5.3.

Этап 3: завершение работы. Как только любой из процессов в контейнере завершается, либо переходит в состояние «FATAL», то вызывается процесс Finisher, который завершает работу Supervisor'a, а вместе с этим завершается и работа контейнера.

5.3 Устройство Test

Наконец, рассмотрим сам процесс тестирования. Тестирующий код написан с использованием фреймворка pytest и его исполнение можно разделить на 3 основных этапа:

Этап 1: развёртывание инфраструктуры. На этом этапе инициализируется логгер, скачивается приложение из Google Play, инициализируется драйвер Android-устройства, поднимается Appium-сервер и создаётся Appium-клиент, а также создаётся объект, предоставляющий высокоуровневый интерфейс для работы с Android-устройством.

Этап 2: исполнение тестового сценария. На данном этапе происходит авторизация в Acronis Cloud, затем происходит авторизация в приложении Acronis Backup, после этого проверяется, что устройство добавлено в Cloud и, в завершение, устройство удаляется из Cloud. Если какой-то из этапов отрабатывает некорректно, то тестирование прекращается и

Этап 3: завершение работы. На этом этапе сохраняется скриншот состояния Android-устройства после исполнения тестового кода, а затем выключается вся используемая инфраструктура: драйвер устройства, логгер, клиент и сервер Appium.

6. Результаты

В рамках данной работы была создана система, позволяющая производить автоматизированное GUI-тестирование Android-приложения Acronis True Image и на основе этой был реализован следующий тестовый сценарий:

- Загрузка последней версии приложения из Google Play
- Установка приложения в мобильном окружении
- Авторизация в приложении
- Проверка, что устройство добавлено в Acronis Cloud

Благодаря реализации функционала для работы с данным приложением, тесты имеют простой для понимания вид. Ниже приведён пример тестового сценария, описанного выше.

```
6
7   class TestAndroid(BaseMobileTest):
8
9       def test_01_sign_in(self):
10          self.logger.info('"Sign in" test started')
11          cloud_client = CloudClient(self.logger)
12          cloud_client.login(cloud_creds)
13          Authorization(self.logger, self.driver).sign_in_to_android(creds=credentials_android)
14          cloud_client.check_device_added()
15          cloud_client.remove_device()
16          self.logger.info('"Sign in" test finished')
17
```

Рисунок 7: код тестового сценария

Кроме того, было подготовлено решение для использования в основном процессе тестирования, а именно:

- Виртуальное окружение было запаковано в docker-контейнер
- Подготовлен код, который по требованию будет разворачивать контейнер из docker-образа и исполнять тестовый сценарий, описанный выше.

Развёртывание и запуск всей системы происходит в 2 команды в терминале (выполняются из-под superuser’а):

- ./build.sh – сборка docker-образа
- ./run.sh – запуск системы

6.1 Пример работы тестового сценария

Рассмотрим пример работы тестового сценария.

Этап 0: сборка контейнера.

Запускаем команду sudo ./build.sh и docker-daemon собирает docker-образ контейнера.

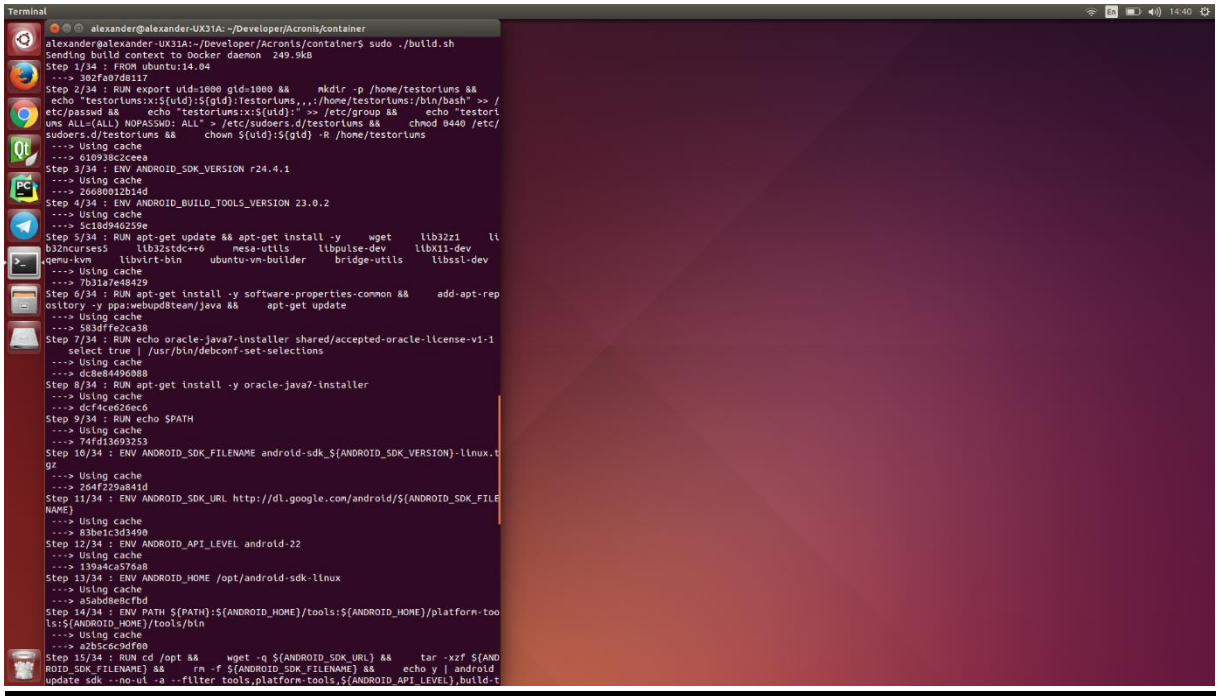

Рисунок 8: сборка docker-образа

Этап 1: запуск эмулятора и приложения Acronis True Image в нём.

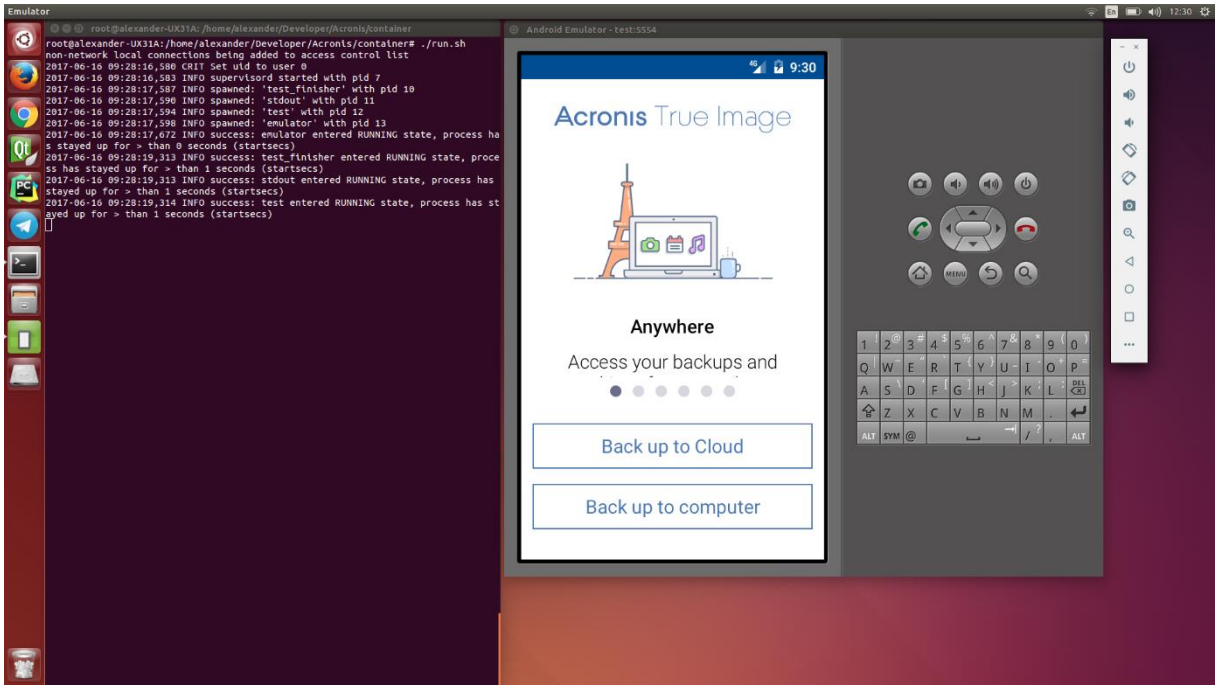

Рисунок 9: запуск эмулятора и приложения

Этап 2: авторизация в Acronis Cloud.

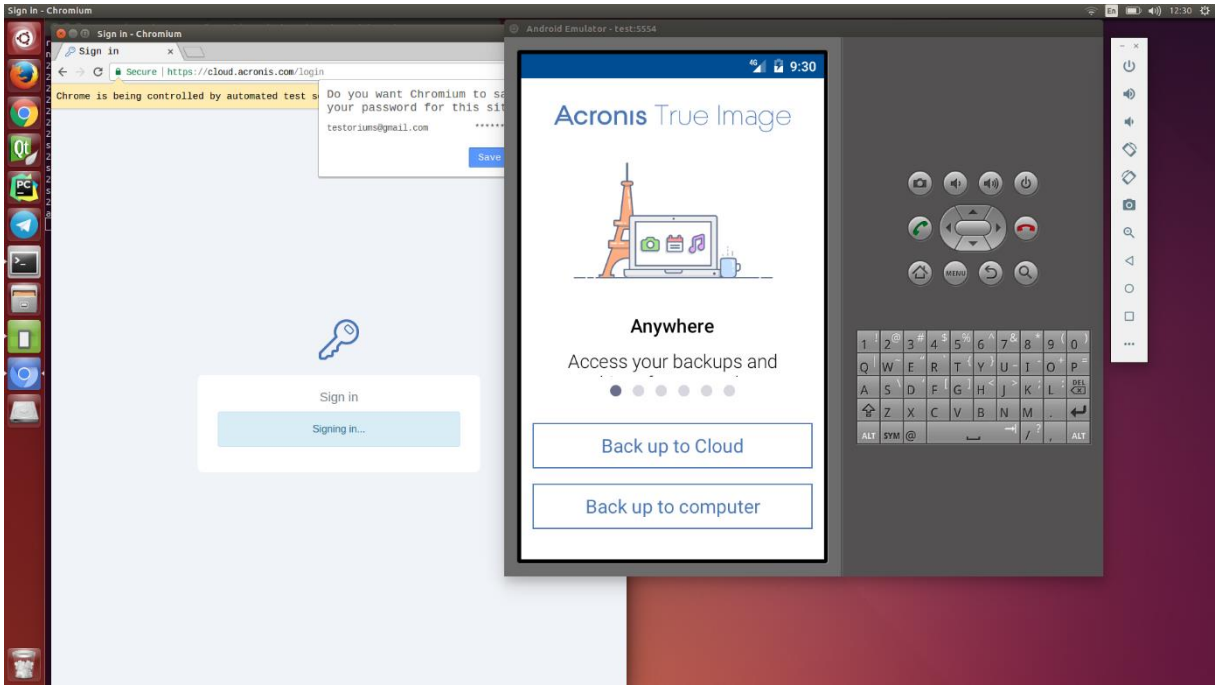

Рисунок 10: авторизация в Acronis Cloud

<u>Этап 3:</u> авторизация в приложении Acronis True Image.

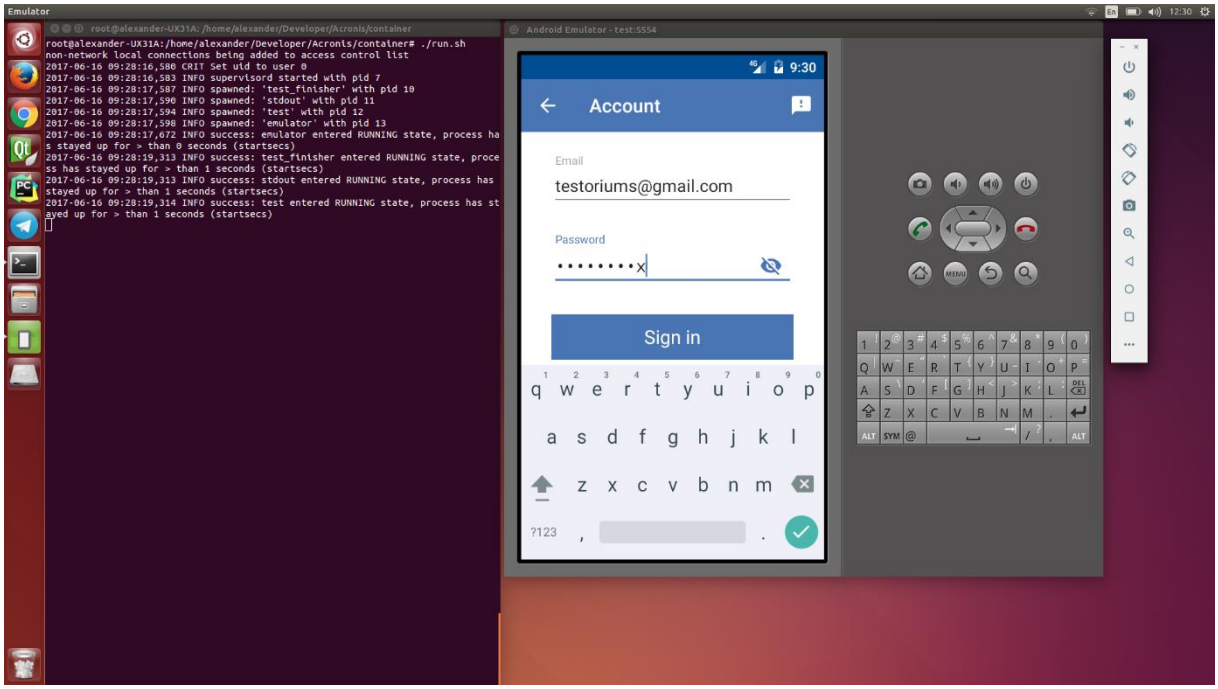

Рисунок 11: ввод данных тестового аккаунта

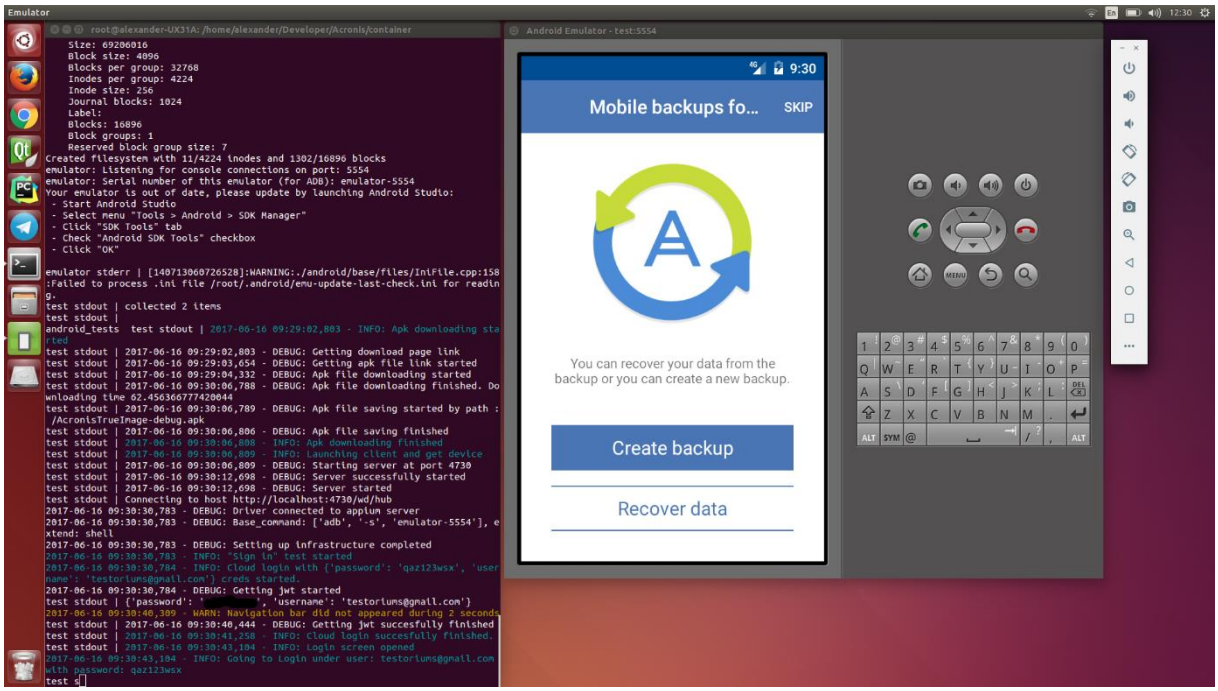

Рисунок 12: авторизация успешно пройдена

Этап 4: выход из приложения.

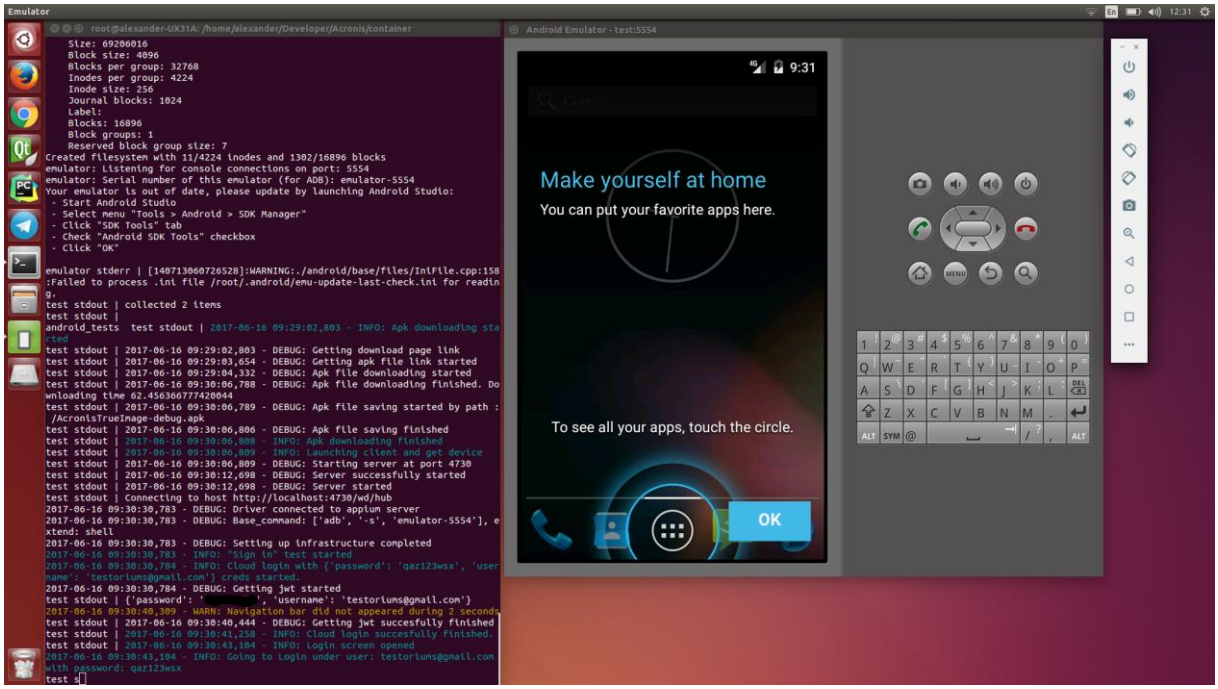

Рисунок 13: выход из приложения

Этап 5: завершение работы системы.

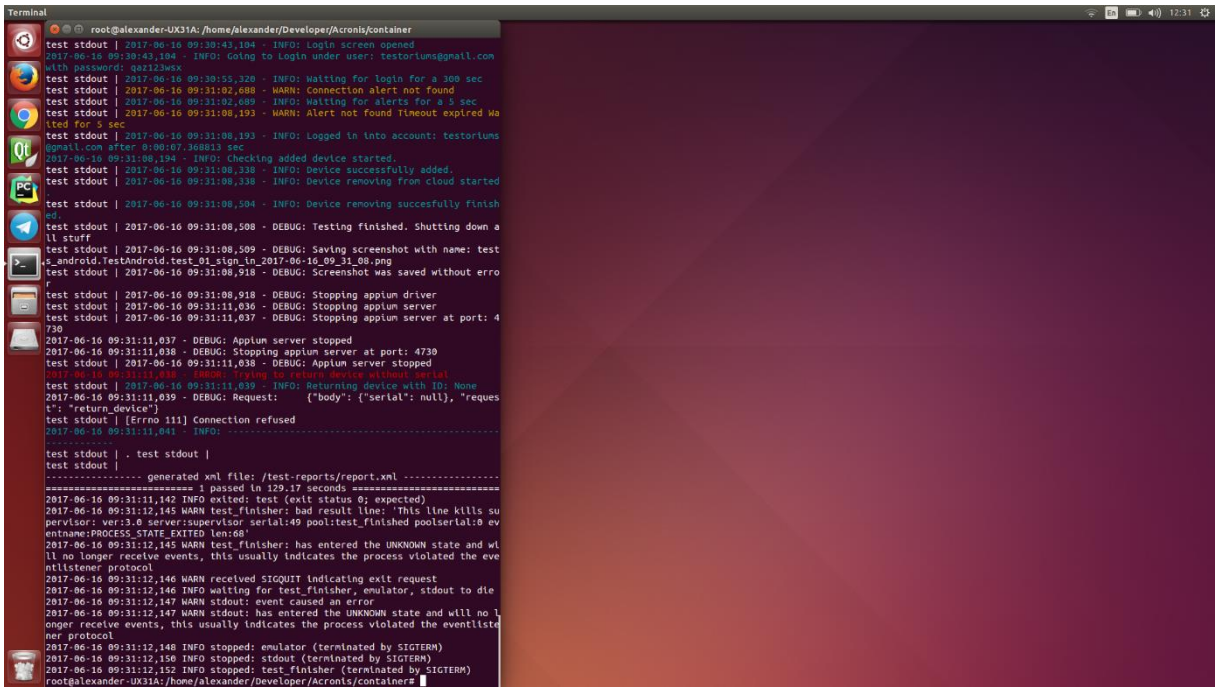

Рисунок 14: завершение работы системы

7. Дальнейшее развитие

Есть два пути дальнейшего развития.

Первый – на основе уже реализованной инфраструктуры проработать High Level Test Design и реализовать полное покрытие автотестами основного функционала продукта Acronis True Image.

Второй – реализовать поддержку тестовой инфраструктурой операционной системы iOS. В данном случае не придётся реализовывать аналогичную инфраструктуру с нуля, потому что, как уже говорилось ранее, Appium поддерживает iOS и можно будет использовать большую часть наработок.

8. Список литературы и ссылки

1. https://ru.wikipedia.org/wiki/Автоматизированное_тестирование
2. Janet Gregory, Lisa Crispin – More Agile Testing: Learning Journeys for the Whole Team – Addison-Wesley Professional, 2014, p. 227
3. https://docs.python.org/3.5/tutorial/index.html
4. http://appium.io/
5. https://wiki.python.org/moin/PyTest
6. https://docs.pytest.org/
7. https://www.slant.co/topics/2621/viewpoints/2/~python-unit-testing-frameworks~pytest
8. https://www.docker.com
9. https://dzone.com/articles/mobile-test-automation-framework-trends
10. https://www.ispe.org/pe-ja/roi-of-test-automation.pdf
11. https://www.ibm.com/developerworks/rational/library/content/RationalEdge/dec02/TestAutomation_TheRationalEdge_Dec2002.pdf
12. http://rasmus.selsmark.dk/post/2012/08/09/Experiences-of-Test-Automation.aspx
13. https://habrahabr.ru/post/189912/
14. https://habrahabr.ru/post/277699/
15. Manoj Hans – Appium Essentials – PACKT, 2015

Приложение 1. Список пакетов

Linux:

ubuntu 14.04

Linux packages:

curl

wget

lib32z1

lib32ncurses5

lib32stdc++6

mesa-utils

libpulse-dev

libX11-dev

qemu-kvm

libvirt-bin

ubuntu-vm-builder

bridge-utils

libssl-dev

Java packages:

software-properties-common

oracle-java7-installer

Appium packages:

nodejs

appium

wd

Python packages:

python3.5

python3-pip

python3-dev

build-essential

python-pip

python3.5-dev

Chrome packages:

chromium-chromedriver

libpango1.0-0

fonts-liberation

libappindicator1

Python 3 pip packages:

pip

virtualenv

pytest

Appium-Python-Client

Pillow

pytest-rerunfailures

retrying

lxml

requests

supervisor-stdout

selenium==3.0.2

jwt

Python 2 pip packages:

supervisor

supervisor-stdout