Building, Testing & Deploying Android Apps with Jenkins

Christopher Orr
iosphere GmbH

June 25, 2014

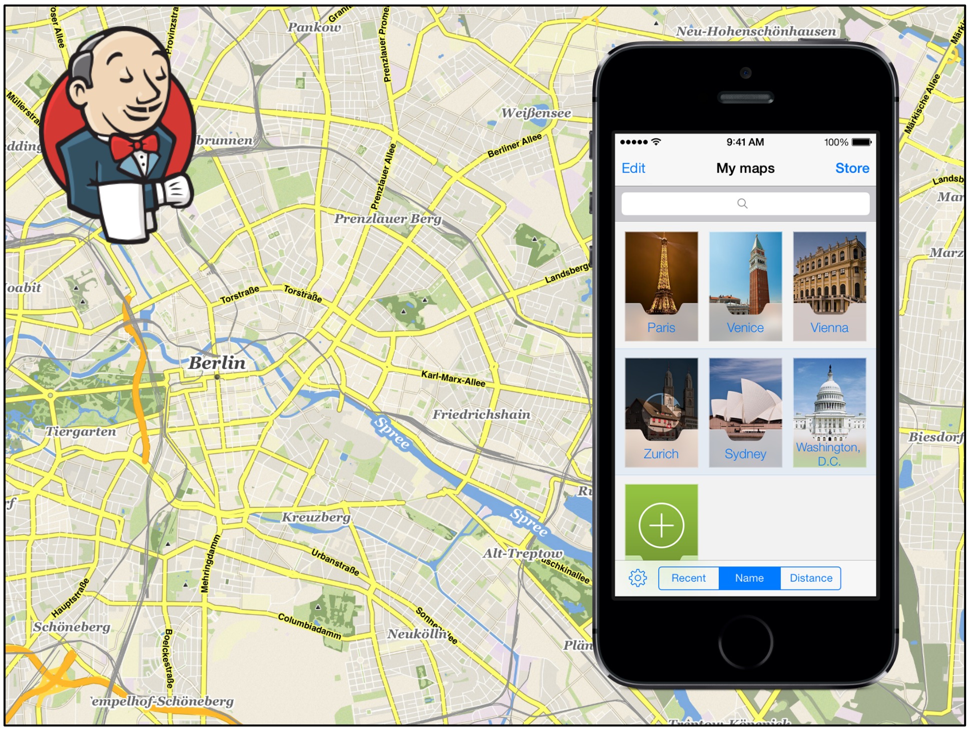
#jenkinsconf

First, a quick bit about what my connection is to Jenkins and to Android...

iosphere

I work in Cologne for a company called iosphere where we specialise in building (award-winning!) mobile applications, as well as often building the supporting backends, admin interfaces etc.

We’re also big fans of Jenkins!



We work on our own products, including OffMaps, which gives you access to detailed street maps, Wikipedia and public transport data, plus helpful stuff like sightseeing tips — all available, including search functionality, without an internet connection.

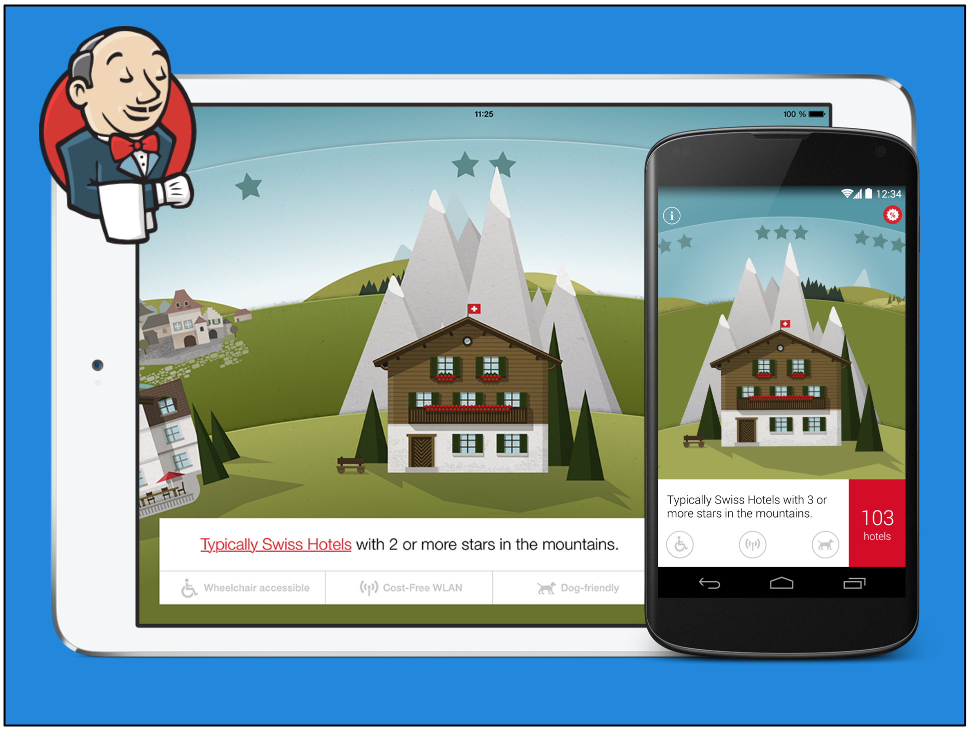
All of this is displayed using our own vector map rendering engine.

The app and dependencies are built by Jenkins and, behind the scenes, a lot of OpenStreetMap data processing, job coordination, data deployments, log management etc. is handled by Jenkins.

In addition to building our own products, we also do work for a variety of customers.

For a mobile payments startup here in Berlin, we do a lot of work building mobile apps for iOS and Android, interfacing with credit card-reading hardware, and we work on firmware development, security and so on.

Again, here we rely heavily on Jenkins for building, testing and deploying the apps.


11:25
100 %
Typically Swiss Hotels with 2 or more stars in the mountains.
Wheelchair accessible
Cost-Free WLAN
Dog-friendly
12:34
Typically Swiss Hotels with 3 or more stars in the mountains.
103
hotels

We love building apps which look great (while of course being functional), in this case a really nice app for finding top hotels in Switzerland.

Of course the apps for all platforms are built and tested by Jenkins, but we also run the hotel data aggregation from Jenkins, and we built a custom frontend for our customer — using some nice web technologies and the Jenkins API — so that they can easily update the dataset used by the app, without actually having to ever access or see the specifics of our Jenkins instance.

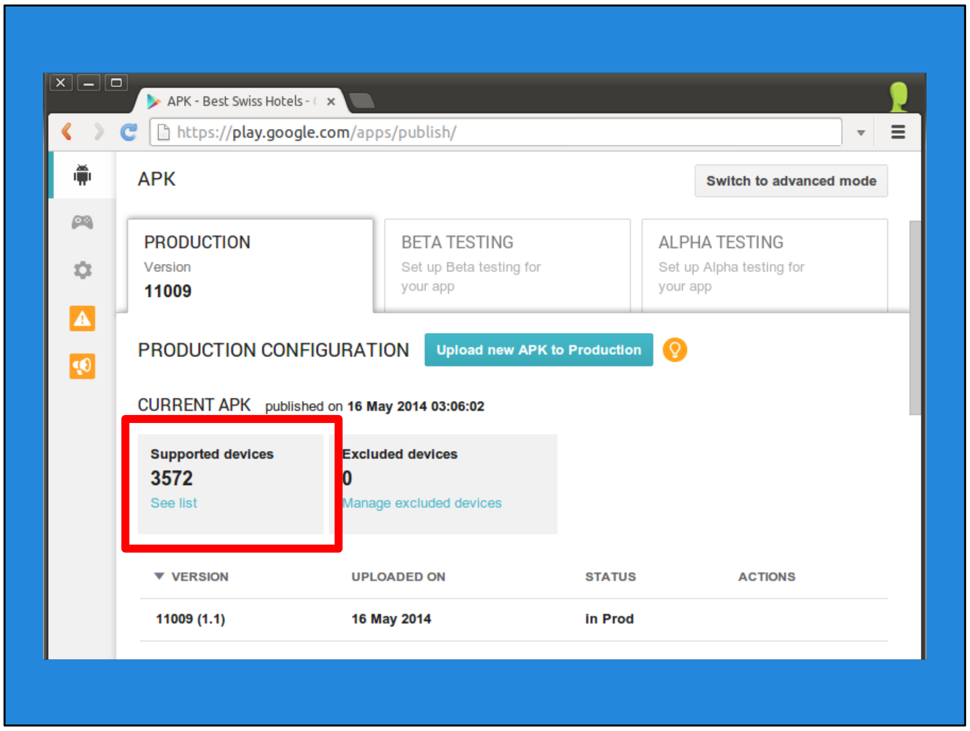
But this talk is about Android — I've been an Android developer for five years now, and it’s what I spend most of my time on at iosphere.

So what are some challenges we face when trying to build high quality Android apps?

The first challenge on the way to developing great Android apps is Android itself.

There is a huge diversity of Android devices — it now runs on smartphones, tablets, TVs, games consoles, watches, vending machines…

But how does this affect us?



APK - Best Swiss Hotels -

https://play.google.com/apps/publish/

APK

Switch to advanced mode

PRODUCTION	BETA TESTING	ALPHA TESTING
Version 11009	Set up Beta testing for your app	Set up Alpha testing for your app

PRODUCTION CONFIGURATION

Upload new APK to Production

CURRENT APK published on 16 May 2014 03:06:02

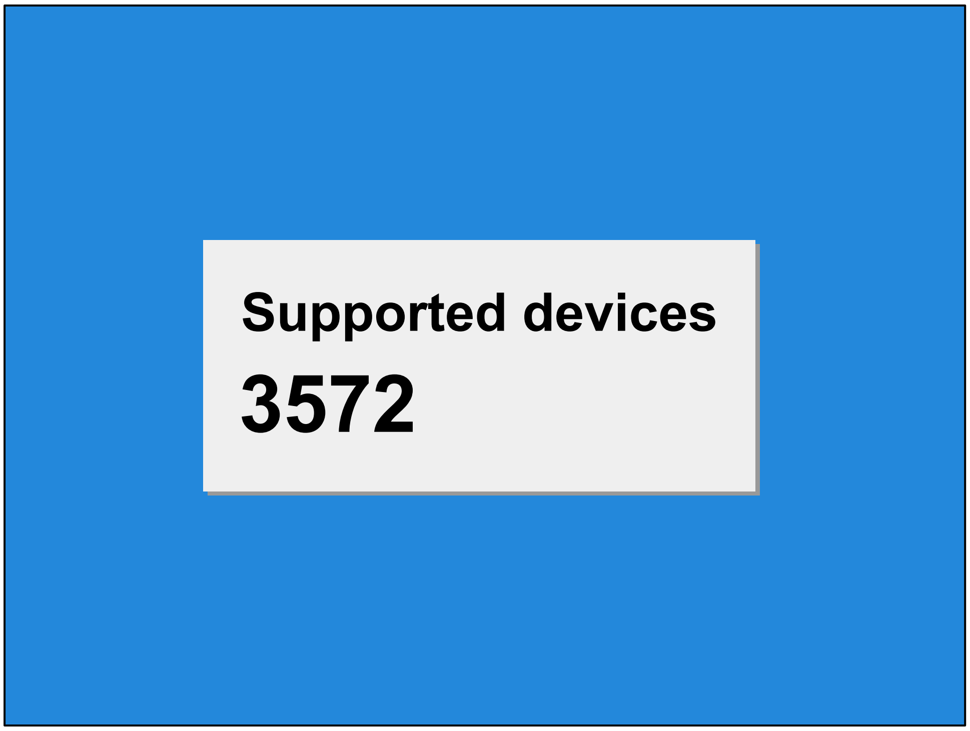
Supported devices	Excluded devices
3572	0
See list	Manage excluded devices

VERSION	UPLOADED ON	STATUS	ACTIONS
11009 (1.1)	16 May 2014	in Prod	

Well, if I go to upload an app to the Google Play store, I see this page.

When I upload my app, and it has certain requirements, e.g. a device that’s running at least Android 4.0, has a touchscreen (which rules out TVs and maybe vending machines), and it needs the Google Maps APIs available.

In this example, we see that there are 3752 different devices which fulfil these criteria.



Supported devices

3572

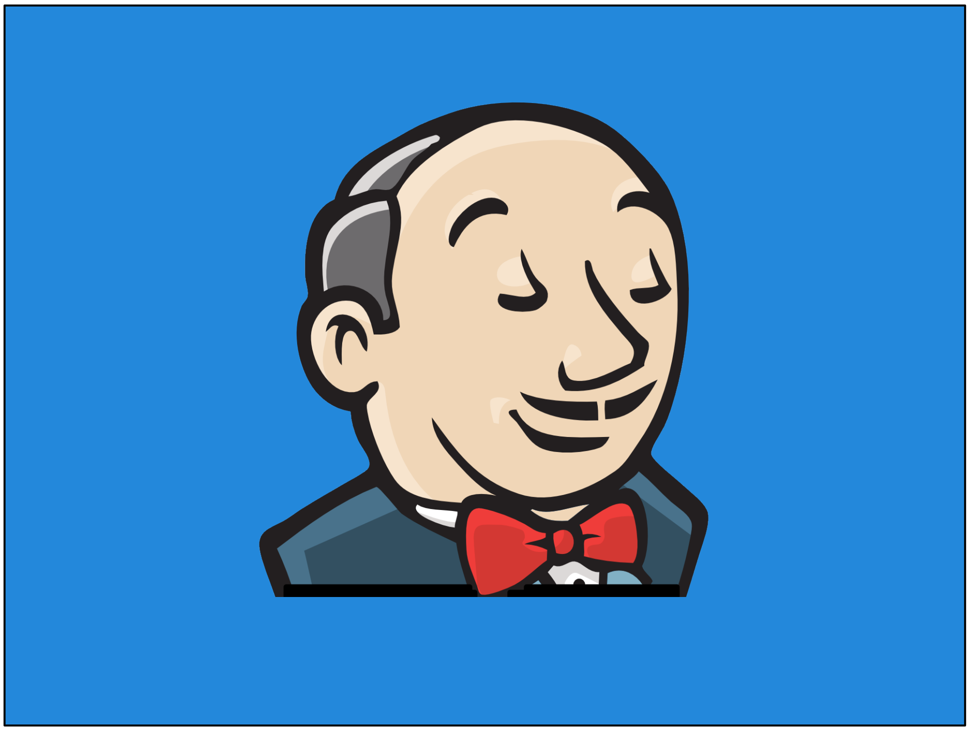
That’s 3572 different device models, with different combinations of OS version, vastly different screen sizes, maybe some with a hardware keyboard, maybe some with only wifi, and perhaps some with only 256MB of RAM, or perhaps with 2GB, and so on.

So while this diversity and choice of hardware is great for us as end users, it’s a pain for developers hoping to provide apps which work well across as many of these as variants as possible.

The other problem we encounter is developers... or the usual software development process.

Writing mobile apps is not hugely different to other types of apps.

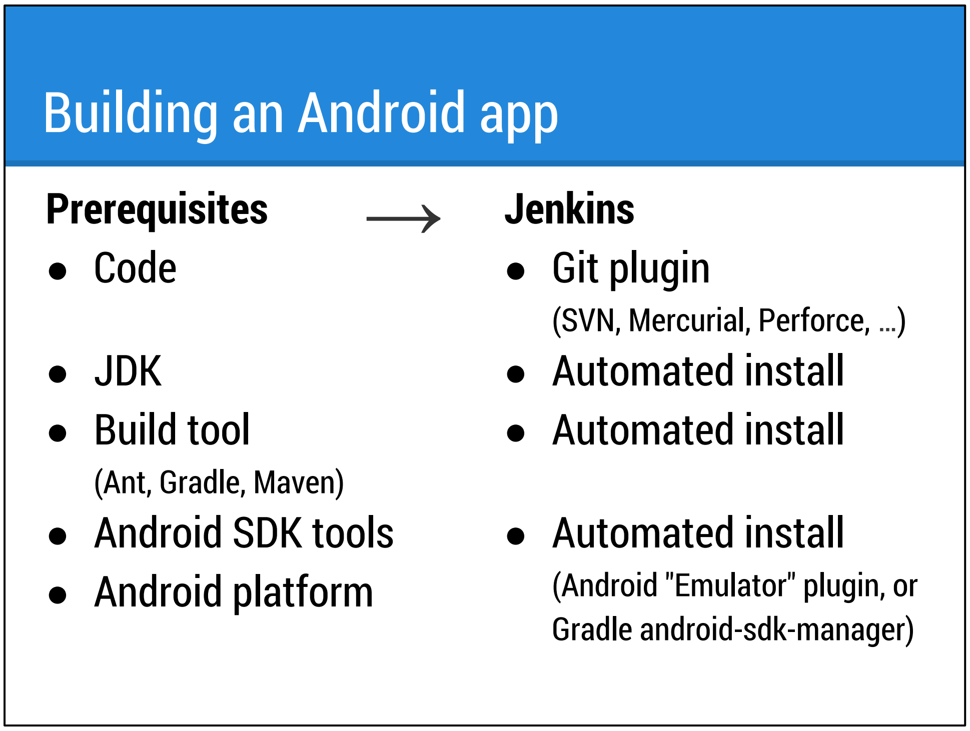
We still need to be able to tackle problems like:
- "Runs on my machine"
- Different developers on the team building against different versions of Ant/Java/Android SDK tools etc.
- Testing with different OS versions and devices



So obviously we know that Jenkins can help with these typical problems, but what exactly are the steps we need to take to go from Android code to happy end users of our app?

The first step with Jenkins is **building an app**.

This gives us the assurance that our code compiles in a clean, independent environment, and gives us a central point to do some static analysis and code quality reporting.



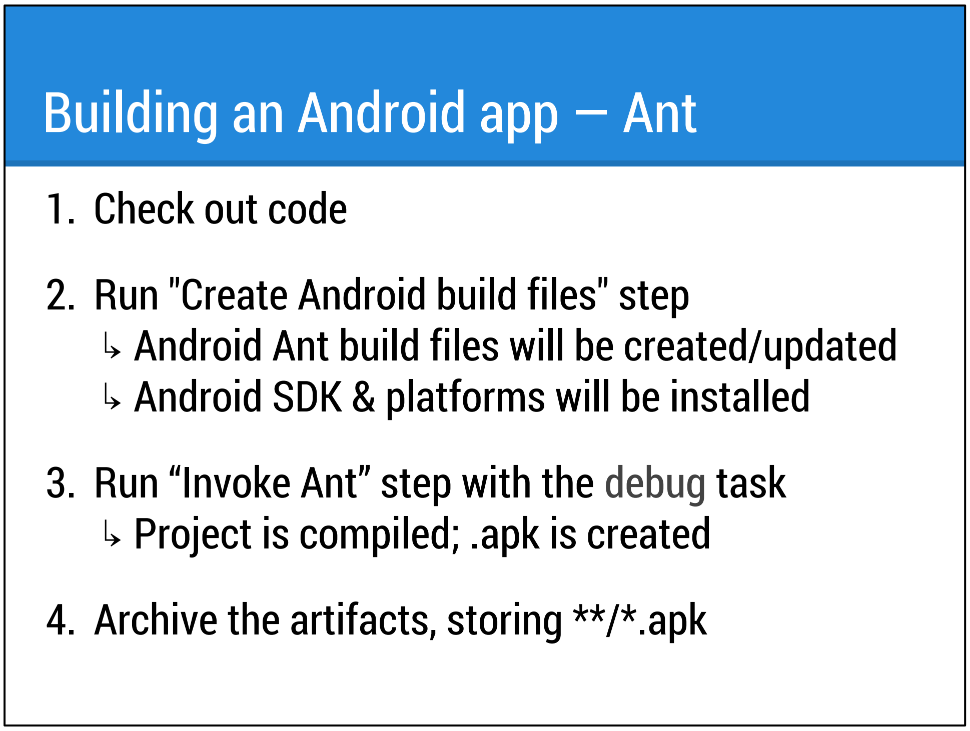
Building an Android app

Prerequisites →	**Jenkins**
● Code	● Git plugin (SVN, Mercurial, Perforce, ...)
● JDK	● Automated install
● Build tool (Ant, Gradle, Maven)	● Automated install
● Android SDK tools	● Automated install (Android "Emulator" plugin, or Gradle android-sdk-manager)
● Android platform	

What do we need to build Android apps, and how can Jenkins help set this up?

Fetching the code is handled by a plethora of Jenkins SCM plugins.
Most prerequisites can be installed automatically by Jenkins — installation of the JDK, Ant or Maven is built into Jenkins.

Finally, the Android Emulator Plugin for Jenkins automates the installation of the Android SDK Tools, and the specific Android platform version you want to compile your app against.



Building an Android app — Ant

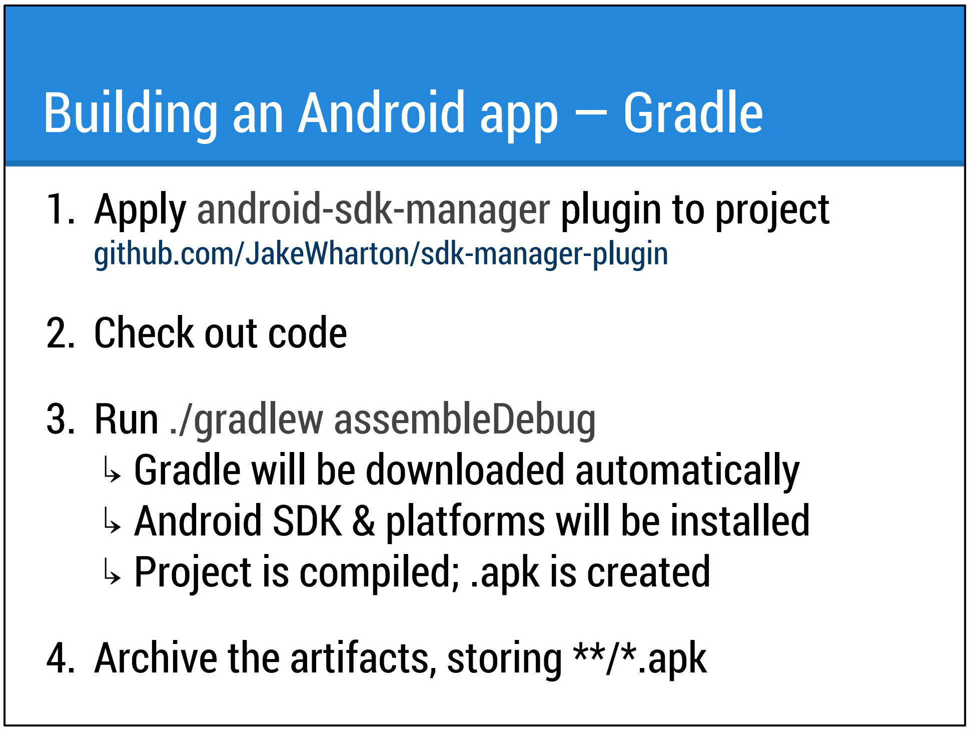
1. Check out code
2. Run "Create Android build files" step
   ↳ Android Ant build files will be created/updated
   ↳ Android SDK & platforms will be installed
3. Run “Invoke Ant” step with the debug task
   ↳ Project is compiled; .apk is created
4. Archive the artifacts, storing **/*.apk

Building an app is relatively straightforward — these are the steps we need to add to a new Jenkins job.

The “Create Android build files” step will scan the files in your job’s workspace, searching for Android project files.
If the Android SDK tools are not yet installed on the Jenkins build machine, this build step will automatically download and install the appropriate SDK version, and download only the required Android platform version required to build this project.
Finally, the build step will update (or create, as necessary) the files required by the Android Ant build system, e.g. build.xml, local.properties etc.

Then you can invoke Jenkins’ built-in Ant built step to compile your project as usual.
If you have a repository which contains two projects: one for your app, and one for your tests, you should run Ant against the “tests/build.xml” file — this will first compile the app APK, and then the test APK.

At the end of your build, you should use the “Archive the artifacts” post-build step, so that we can use the APK files compiled in a later Jenkins job.



Building an Android app — Gradle

1. Apply android-sdk-manager plugin to project
   github.com/JakeWharton/sdk-manager-plugin

2. Check out code

3. Run ./gradlew assembleDebug
   ↳ Gradle will be downloaded automatically
   ↳ Android SDK & platforms will be installed
   ↳ Project is compiled; .apk is created

4. Archive the artifacts, storing **/*.apk

With the newer Android Gradle build system, the build process is even simpler.

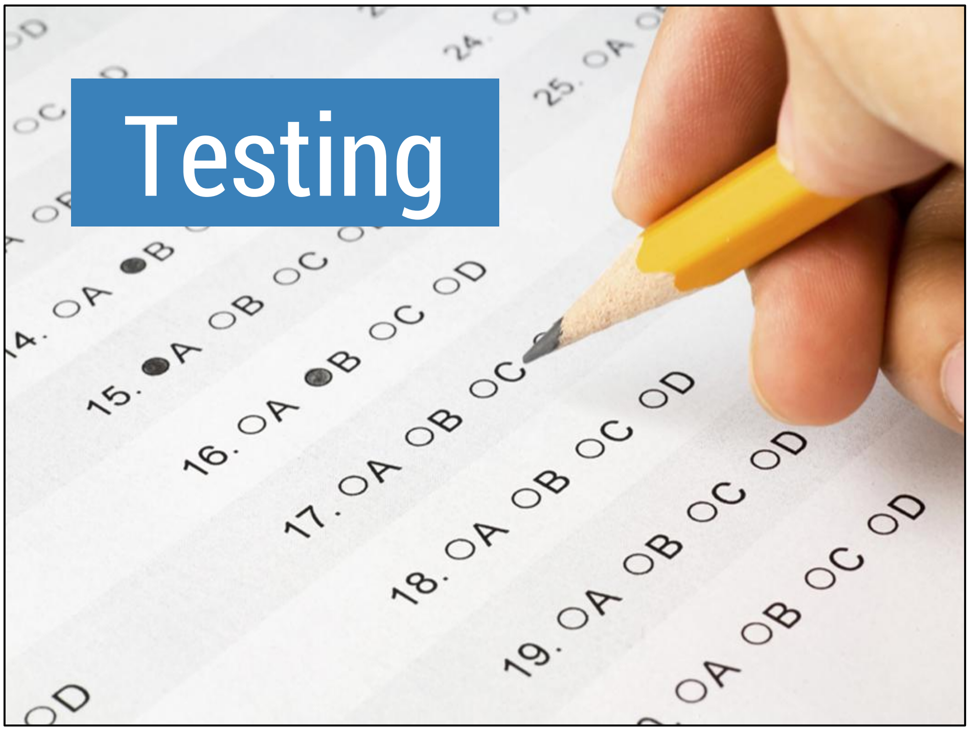
By applying the Android SDK Manager Gradle plugin from Jake Wharton to the project, we don't need to tell Jenkins to setup and install the SDK — the Gradle plugin handles this for us.

i.e. It's essentially as simple as checking out the code and running the Gradle compile command.

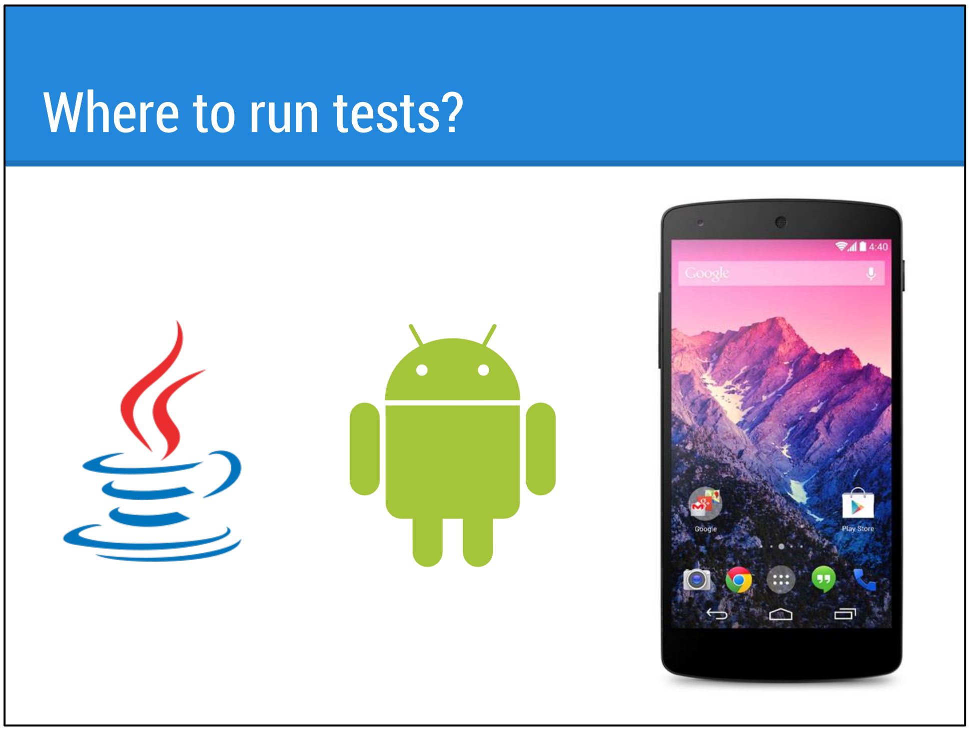
At the end of such a build, we can also run the Lint tool from the Android SDK to do some static analysis of the app, parsing the result with the Android Lint Plugin for Jenkins, allowing us to potentially mark the build as unstable or failed, if Lint reports that a certain number of programming errors were introduced in the latest commit.

Demo

Testing



Now that we have Jenkins building our app, we can think about automating testing of the app.



Where to run tests?

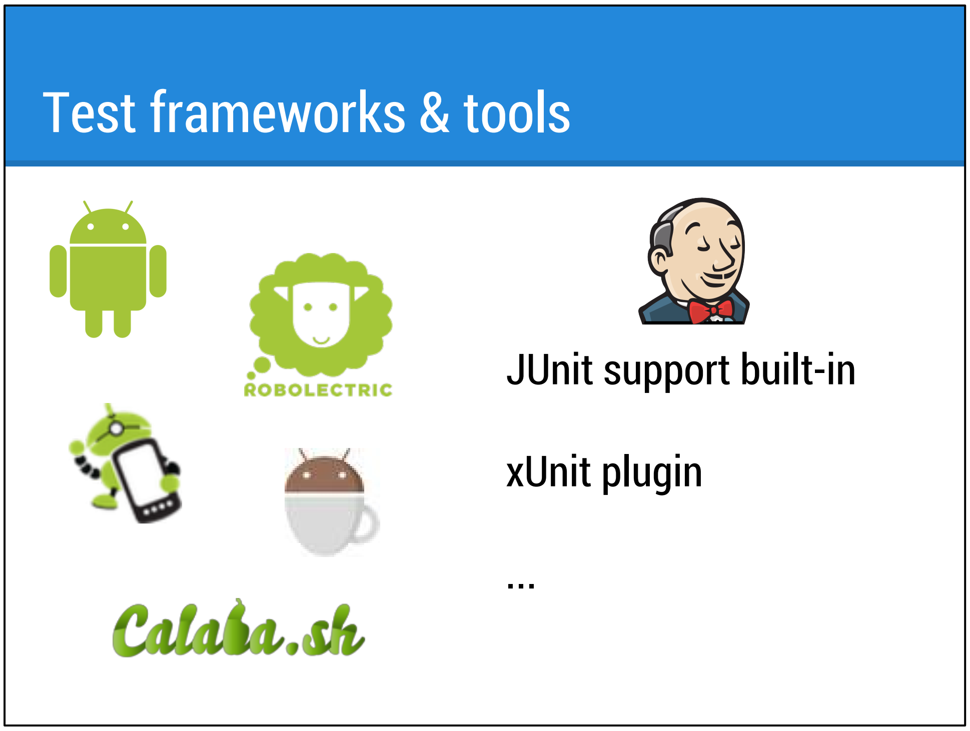
There are several ways in which you can execute automated tests for Android, each with their own trade-offs.

The fastest way to test is to run on the JVM:
- If you have pure Java unit tests, e.g. business logic that doesn’t use Android APIs
- There is also the excellent Robolectric project, which shadows a lot of the Android API, letting you test your app very quickly, without having to package it and install on an emulator or device
  - But the fidelity isn’t there, there can be bugs, and it isn’t an official Android tool

You can start an Android emulator and run your JUnit tests there:
- This allows you to easily test different OS versions, screen sizes, languages, hardware configurations (RAM, GPS etc.)
- Emulators can be slow to start, and still don’t offer full fidelity to a real device

Or you can let your automated tests run on a device:
- You can’t get more realistic than a real device, but buying them can be costly
- Requires dedicated devices, always plugged in to your Jenkins instance, and the Android Debug Bridge (adb) connection and USB can be pretty unstable
- It can be tough to keep devices always in a pristine state, so that each test run is consistent

But all are possible from Jenkins, though we’ll concentrate here on using emulators.


ROBOLECTRIC
Calaba.sh

Test frameworks & tools

JUnit support built-in

xUnit plugin

...

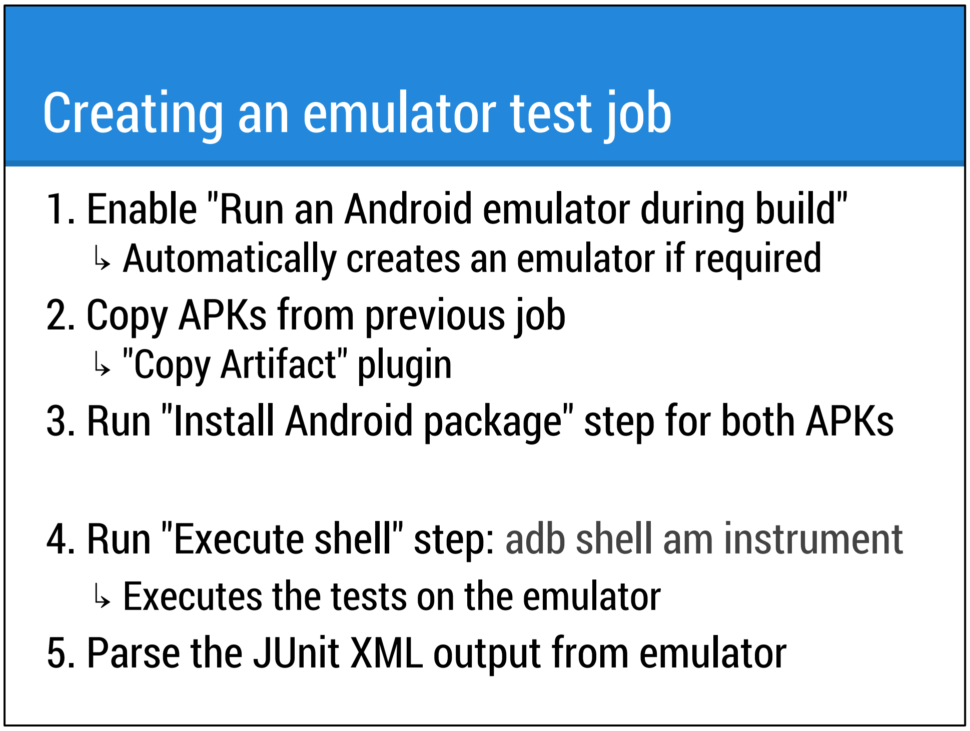
Regardless of which approach or test framework we choose, we first need to write — and then execute — the tests.
The built-in Android test framework and projects like Robolectric or Espresso are all running under JUnit3 or JUnit4 test runners.

So how do these fit into Jenkins?
Jenkins' built-in test reporting framework supports JUnit XML files, and using the xUnit and similar plugins, you can report on differing file formats.

The default Android test running unfortunately does not output JUnit XML, but a rather information-free format.
The new Gradle build system is better, producing JUnit XML in the expected format.

But for the former, or for those who want to run tests without the Gradle build script, there is an open source project which can help:
https://github.com/jsankey/android-junit-report

This will write a JUnit XML file to the emulator or device as the tests run, which you can then fetch and parse.



Creating an emulator test job

1. Enable "Run an Android emulator during build"
   ↳ Automatically creates an emulator if required
2. Copy APKs from previous job
   ↳ "Copy Artifact" plugin
3. Run "Install Android package" step for both APKs

4. Run "Execute shell" step: adb shell am instrument
   ↳ Executes the tests on the emulator
5. Parse the JUnit XML output from emulator

These are the steps we need to add to a new Jenkins job to test an app.

Firstly, the "Run an Android emulator during build" option will create an emulator with the properties we specify in the job configuration, with the main properties being Android OS version, locale and screen size/density.
With this option enabled, Jenkins will run the emulator during the build, wait for it to start, and archive the entire emulator log file to disk, shutting the emulator down once the build is complete.

Once the emulator has started, we use the "Copy Artifact" plugin to copy the app and test APKs we built earlier into our workspace. Then we can use two "Install Android package" build steps to install the two APKs onto the running emulator.

Then we can use the Android emulator shell to start the tests running, with the "am instrument" command as documented here: https://developer.android.com/reference/android/test/InstrumentationTestRunner.html

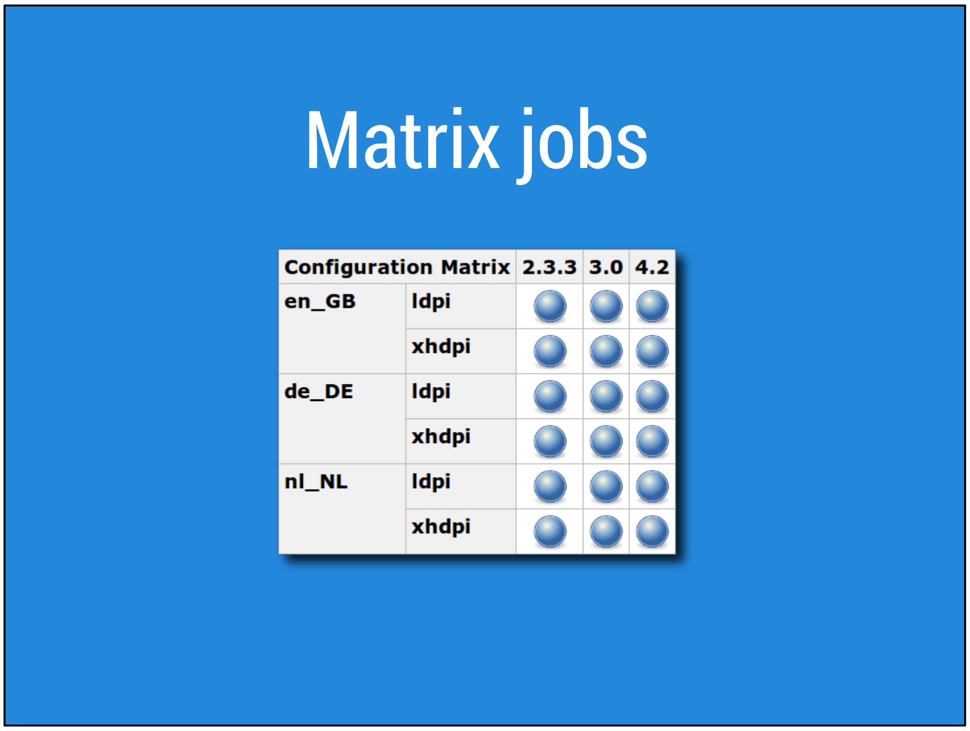
An example, with the JUnit test runner mentioned previously, would be:
```
adb shell am instrument -w com.example.myapp.test/com.zutubi.android.
junitreport.JUnitReportTestRunner
```

Then we can extract the generated JUnit XML report from the emulator:
```
adb pull /data/data/com.example.myapp/files/junit-report.xml .
```

Finally, the “Publish JUnit test result report” post-build action is used to parse the “junit-report.xml” file.

Now, soon after having pushed our code, Jenkins will build and test our app, providing us with test results, and the build will be marked as successful or unstable, depending on whether the tests pass or fail — of course, this can then trigger emails or other notifications to the developers involved, notifying that they need to fix the build...

But how does having this this help with those 3500+ different device models?



Matrix jobs

Configuration Matrix		2.3.3	3.0	4.2
en_GB	ldpi	●	●	●
	xhdpi	●	●	●
de_DE	ldpi	●	●	●
	xhdpi	●	●	●
nl_NL	ldpi	●	●	●
	xhdpi	●	●	●

We can use the power of the “matrix” or “multi-configuration” job type in Jenkins.

This allows us to run the same basic job configuration multiple times, but each time with one variable changed, until all combinations have run.

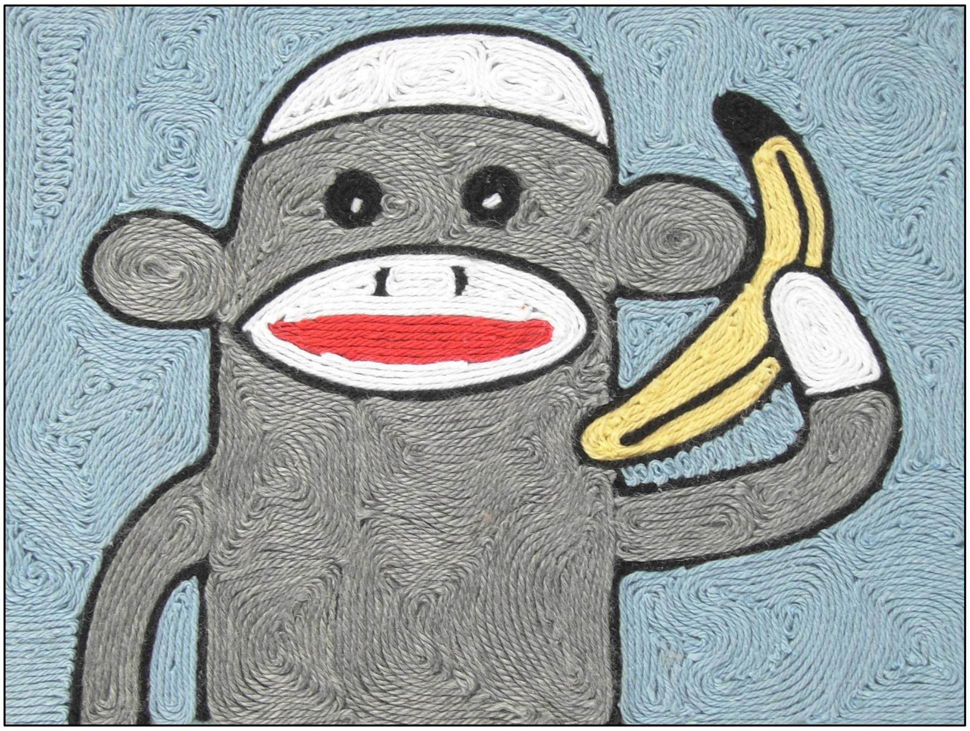
The variables important to us as Android developers are:
- Android version
- Screen size or density
- Locale

While this doesn’t equal 3500 different device models, it covers the majority of the combinations in use, in the real world.

Normally, I would test at least test the *minimum* Android version my app supports, as well as the *target* version.
I would then add the various languages my app supports, and if I have a lot of UI layout differences between screen sizes, e.g. phone-sized and larger tablet-sized layouts, I might add some different screen resolutions.
Decide on which values are important to you, or check your user usage statistics to see which are the most important combinations to cover for your userbase.

The advantage of running your tests across these combinations is that you can find subtle bugs which are specific to a particular screen size, language or OS version.

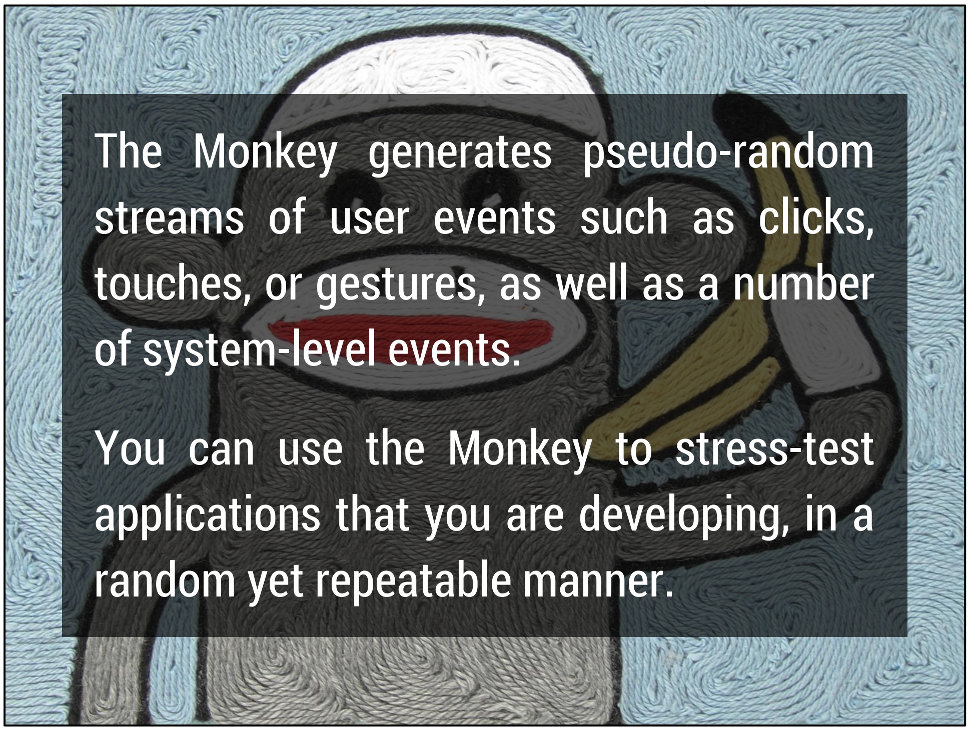
Demo



We are now testing our app on lots of different device combinations — but usually the app is developed and tested in debug mode rather than release mode, as doing so is much faster, certainly during development.

But your release build may contain subtle errors caused by bytecode optimisation or obfuscation if you're using tools like ProGuard. If you're not constantly testing your release version, then it's quite possible that the first people to find errors in your release build could be end users.
Plus in general, there are likely edge cases in your app where a crash can happen after a certain sequence of user input, or after a screen rotation, etc.

We can use the Android SDK tool called Monkey to stress test our apps and find these types of issues.



The Monkey generates pseudo-random streams of user events such as clicks, touches, or gestures, as well as a number of system-level events.

You can use the Monkey to stress-test applications that you are developing, in a random yet repeatable manner.

As described, Monkey sends a stream of touch events, screen rotations, etc to your application.
It generally does this as fast as it can, trying to break your app via making unexpected sequences of input.

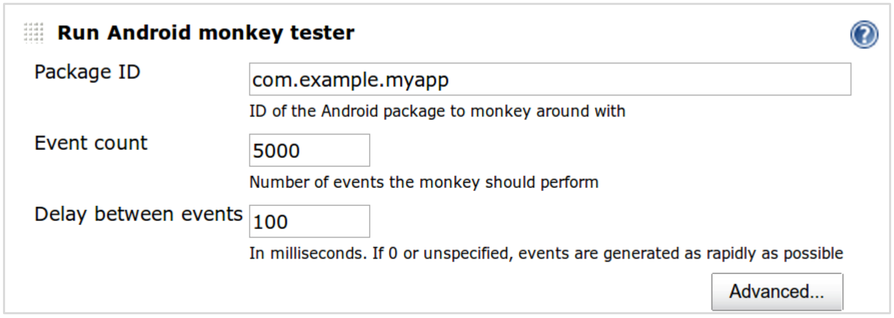
How can we integrate this with our Jenkins setup?

Stress testing with monkey

Enable “Poll SCM” — @midnight

↳ Job only runs if changes were made to the app today

Install APK built in release mode, with ProGuard etc.



Run Android monkey tester

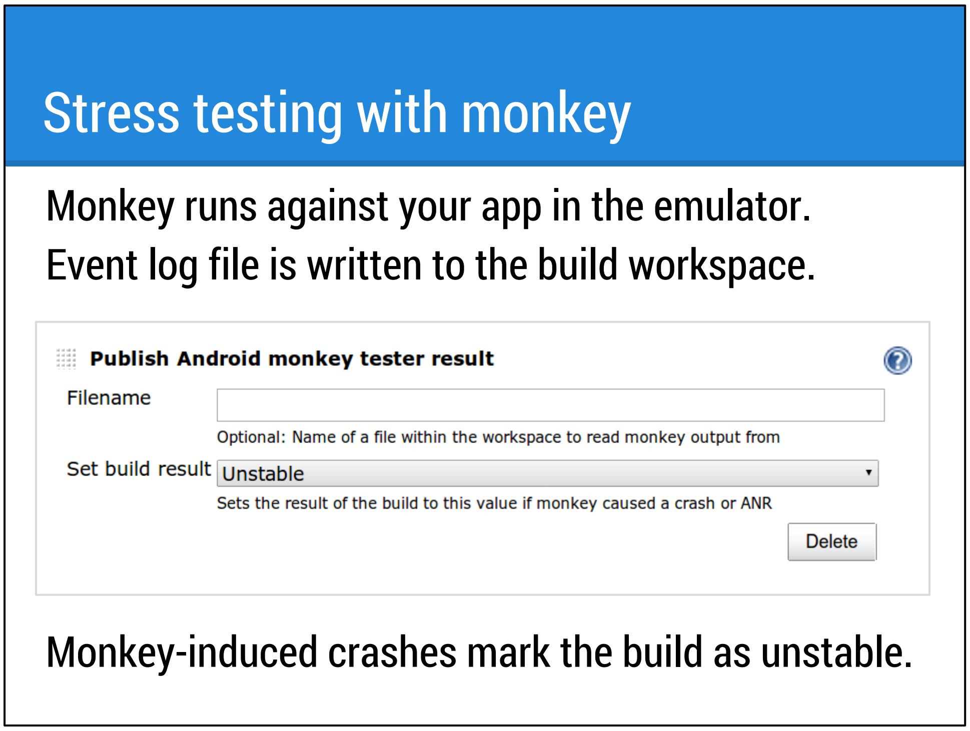
Package ID	com.example.myapp
	ID of the Android package to monkey around with
Event count	5000
	Number of events the monkey should perform
Delay between events	100
	In milliseconds. If 0 or unspecified, events are generated as rapidly as possible

Depending on how many events you ask Monkey to execute, a test run can take from seconds to multiple hours.
It makes more sense to use more events, say 50000 or more, to ensure that Monkey reaches as many parts of your app as possible, in as many scenarios as possible.

As this can take a long time to run, it’s a good idea to run this type of test just once a day, rather than after every commit.

We can ensure that a Jenkins job runs nightly — but only if the app was changed during the day — by enabling “Poll SCM” (as opposed to “Build periodically”) with a value like “@daily” or “@midnight”.

For this stress test job, you want to start an Android emulator, build your APK in release mode, install the APK, and then add the “Run Android monkey tester” build step.



Stress testing with monkey

Monkey runs against your app in the emulator.
Event log file is written to the build workspace.

Publish Android monkey tester result

Filename:

Optional: Name of a file within the workspace to read monkey output from

Set build result: Unstable

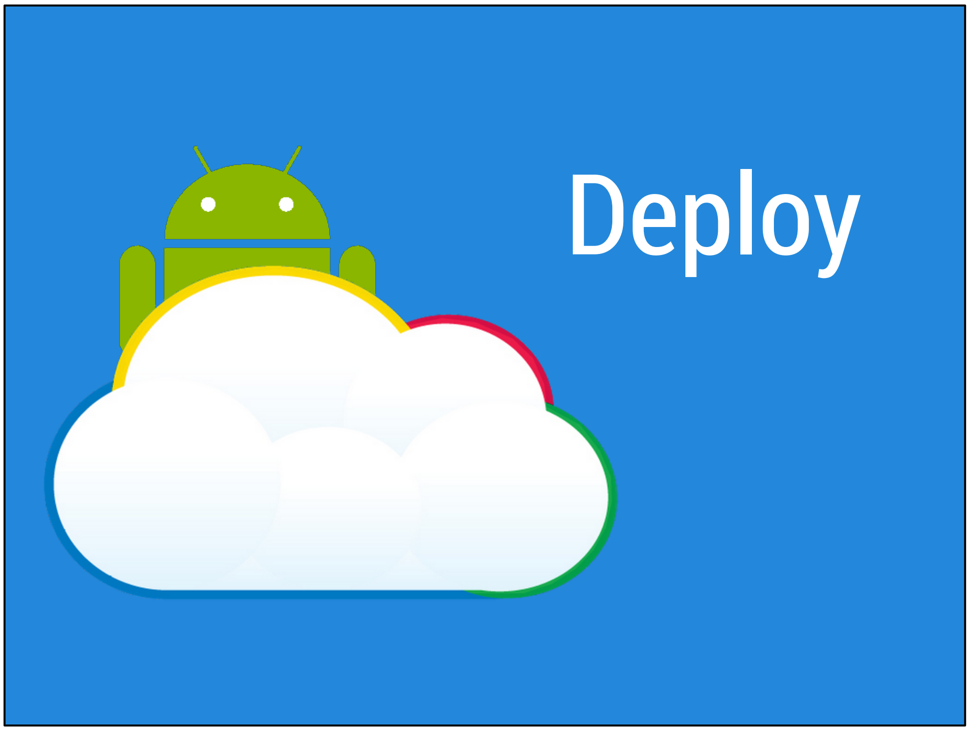
Sets the result of the build to this value if monkey caused a crash or ANR

Monkey-induced crashes mark the build as unstable.

The Monkey tool will then run until the given number of events have been completed, or until the app crashes.

By default, a log of all the events generated and executed by Monkey is written to “monkey.txt” in the workspace.
The “Publish Android monkey tester result” post-build step can parse this file and automatically change the Jenkins build result to unstable or failed, if any crashes occurred.

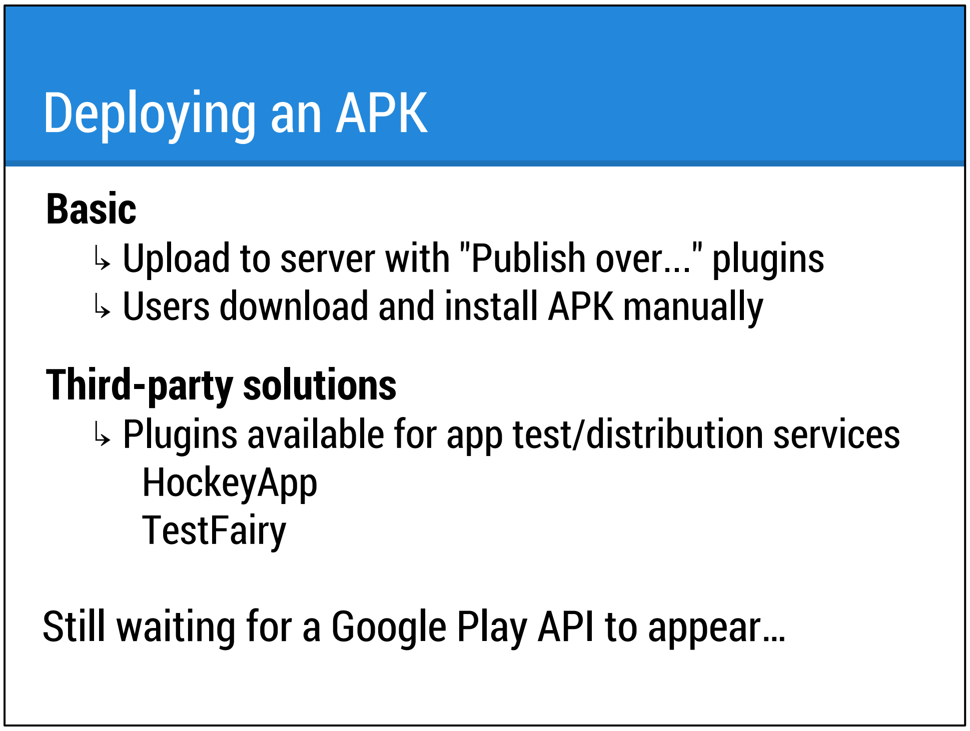
You can also archive the “monkey.txt” file for later inspection.



Deploy

So with the app working on various devices, and stress tested, we’re ready for launch!

How can we get the app into the hands of users, like your QA department, curious product managers or beta testers?



Deploying an APK

Basic

↳ Upload to server with "Publish over..." plugins
↳ Users download and install APK manually

Third-party solutions

↳ Plugins available for app test/distribution services
  HockeyApp
  TestFairy

Still waiting for a Google Play API to appear…

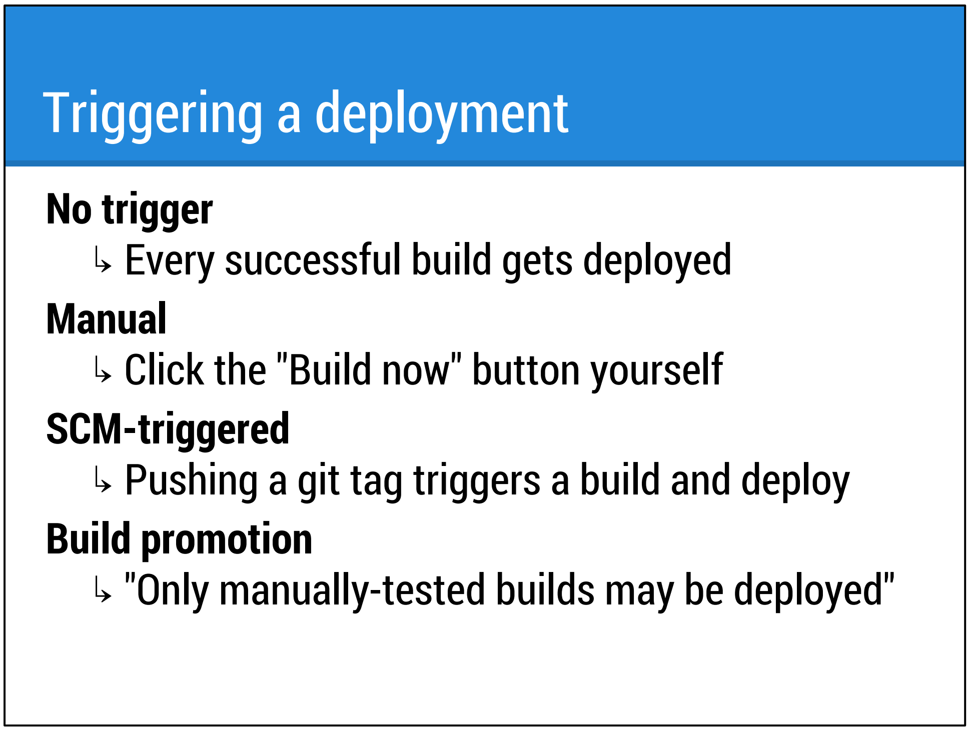
There are various ways we can distribute an APK to end users with Jenkins.

The simplest is to upload the APK to a web server, from which users can manually download and install the app. This is easy to do from Jenkins as there are many plugins for uploading to various types of servers via various protocols, but this doesn't make downloading and installing as simple as it could be for end users.

There are third-party solutions like HockeyApp, which have Jenkins plugins allowing you to upload an app to their service, from where they notify your beta testers that there is a new version available. HockeyApp then gives you reports on how many downloads your app has had, and whether there have been any crashes.

Google Play also offers a similar beta testing service, integrated into the Google Play store, but this is very slow — it takes several hours after uploading an app before users can download it — and as yet there is no support for automatically uploading APKs to the service.
However, a "Google Play Developer Publishing API" was announced at Google I/O 2014, but at this time it's in a closed beta phase.

Once such an API becomes public, it's likely that Jenkins integration will follow, so that building an app and delivering it to test users can be 100% automated.


Triggering a deployment

No trigger

↳ Every successful build gets deployed

Manual

↳ Click the "Build now" button yourself

SCM-triggered

↳ Pushing a git tag triggers a build and deploy

Build promotion

↳ "Only manually-tested builds may be deployed"

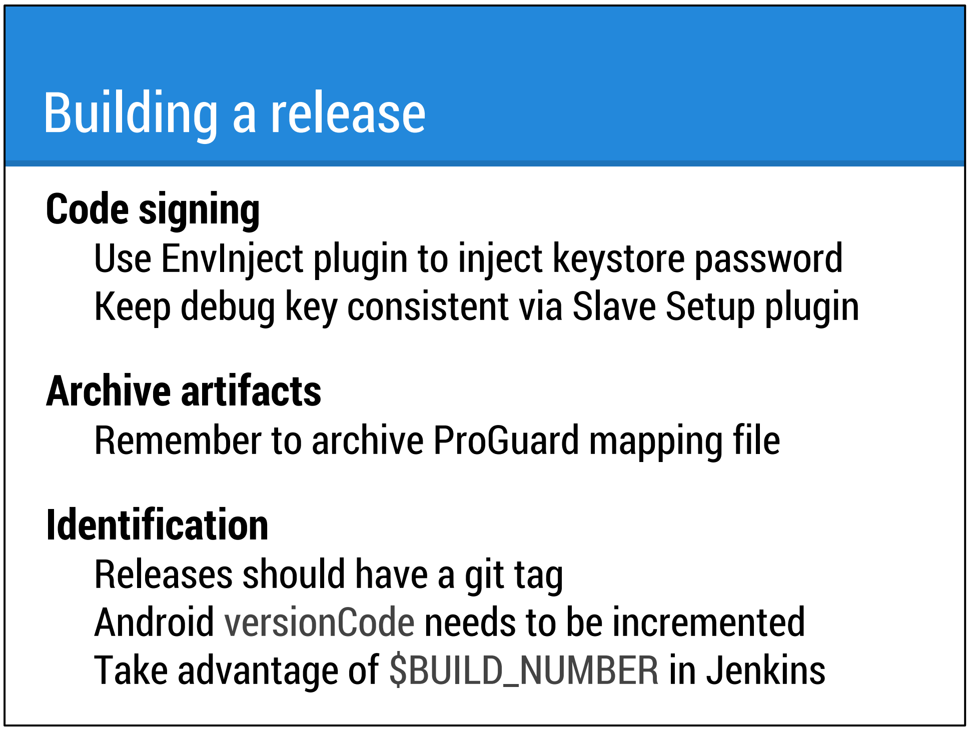
So we know how to deploy our app, but when are we able to do this?

We can decide that every successful build gets uploaded — perhaps useful for alpha testers or other developers on the team.
Similarly, we can trigger builds manually by hitting “Build now” on our release build job from the Jenkins UI.

The approach we use most commonly at iosphere is to trigger release builds via SCM — in our case, we use specially-named git tags to kick off various builds.
For example, we can tag a build as “alpha/123” — regardless of which branch the tagged commit is on; though we usually tag builds from our “develop” branch, as we tend to use the “git flow” model — and we have an “Alpha release” job in Jenkins which will only build new instances of “alpha/*” tags for a given project.

We have a near-identical setup for beta builds — any tag starting with “beta/” will be noticed by Jenkins, have a release app built and uploaded to HockeyApp, ready for distribution to beta testers.

Finally, there is a Jenkins plugin called “Promoted Builds” which allows you to add various approval criteria which must be fulfilled before executing further build steps — for example, a tester has to manually click “Approve” after manually testing an app before the build is considered ready (or “promoted”) to be placed into production or beta testing.



Building a release

Code signing

Use EnvInject plugin to inject keystore password
Keep debug key consistent via Slave Setup plugin

Archive artifacts

Remember to archive ProGuard mapping file

Identification

Releases should have a git tag
Android versionCode needs to be incremented
Take advantage of $BUILD_NUMBER in Jenkins

What should we consider when building a release of an Android app?

We need to know the password for the release keystore that the app will be signed with. It's critical that this is kept a secret, so of course this password should not be hardcoded into any build scripts, nor checked into our source repository.
So the EnvInject plugin comes in handy — we can use the "Inject passwords to the build as environment variables" option to make the signing password available during a build, allowing the Ant or Gradle build can use it, while keeping the value encrypted on disk in the Jenkins config, and out of the app's source repository.

For non-release builds, it can be handy to use a consistent signing key between all Jenkins slaves — by default the Android SDK generates a random key for every machine if a key does not already exist. By selecting one key and using the Slave Setup plugin to ensure all slaves have the same key (and perhaps sharing the same key between developers), it means there will be no more "INSTALL_PARSE_FAILED_INCONSISTENT_CERTIFICATES" errors.

If you're using ProGuard to obfuscate your code, you should archive the generated mapping text file, so that later you can use the retrace tool to decode obfuscated stack traces.

A release build should have some traceability.
You should know which git tag, and ideally which Jenkins build it came from, if you need to track down possible build issues later.

As mentioned, we tag all of our alpha and beta builds with specially-named git tags, so that they're built specially by Jenkins — the same goes for release builds.

Each Android release needs to have a unique, increasing version code value. We can use the Jenkins $BUILD_NUMBER environment variable which is available during every built to bake this value into our APK. The Gradle build system makes this easy.

```
ext.vMajor = 1
ext.vMinor = 1

android {
  defaultConfig {
    versionName computeVersionName()
    versionCode computeVersionCode()
  }
}

// Returns “1.1”
def computeVersionName() {
  return String.format('%d.%d', vMajor, vMinor)
}

// Returns 11009, for Jenkins build #9
def computeVersionCode() {
  return (vMajor * 10000) + (vMinor * 1000) +
    Integer.valueOf(System.env.BUILD_NUMBER ?: 0)
}
```

This is an extract from a build.gradle file I use.
Rather than defining the “versionCode” and “versionName” values in the AndroidManifest.xml file, and having to increase them manually for every new version published, I generate these values at build time.

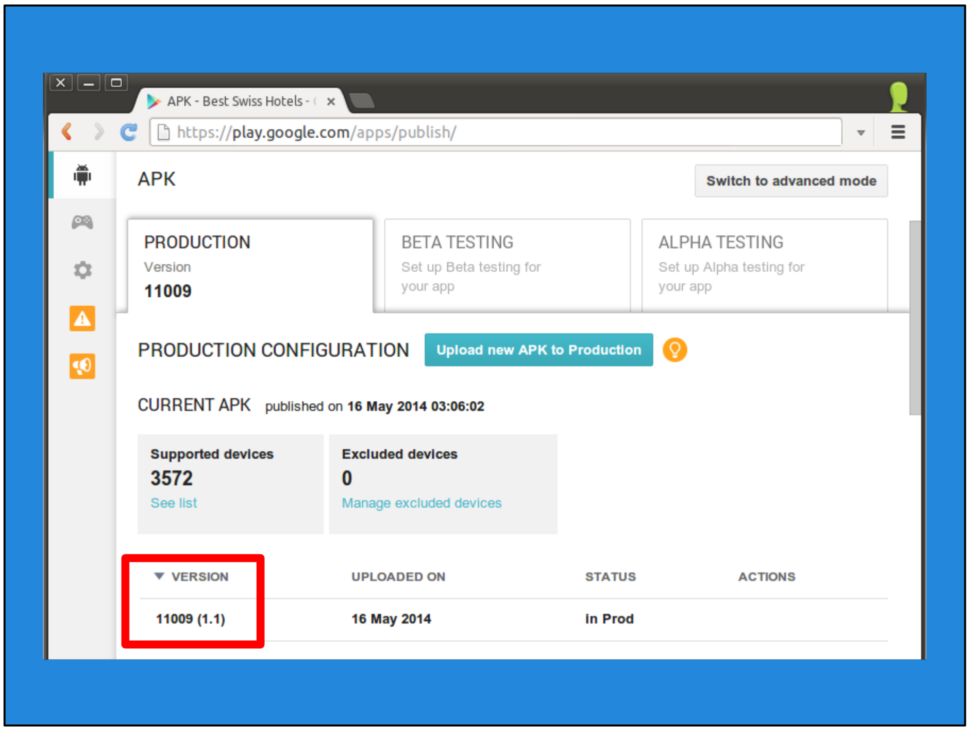
At the top, the major and minor variables represent release “1.1”, and this String value is what the computeVersionName() function will return.

At the bottom, the computeVersionCode() method uses a combination of the major and minor version with the Jenkins build number.
In this example, if the APK is built during build #9 of our job, the APK will have the versionCode 11009.
If the build is not being run from a local developer machine, rather than by Jenkins, the BUILD_NUMBER variable will not be set and the digit at the end will simply be zero.

```
buildTypes {
  debug {
    applicationIdSuffix '.dev'
  }
  beta {
    num = System.env.BUILD_NUMBER
    versionNameSuffix 'b' + num
    applicationIdSuffix '.beta'
    ...
  }
  ...
}
```

So while the automatically-incremented versionCode gives the developers some traceability about when a particular APK was built and what it contains, it doesn't necessarily help when asking app users exactly which app build they're using.

This is another Gradle build config snippet we use, in which the "beta" versions of our app have a suffix applied to their versionName — the user-visible versionvalue.
For version 1.1 beta, built by Jenkins job #9, the versionName would have the value "1.1b9" — and we display this value on the "About" screen of our app, so we can easily ask beta testers which version they're using.
From there, we can see which Jenkins build created the APK in use, and from there we can see the exact git commit that was built.



APK - Best Swiss Hotels

https://play.google.com/apps/publish/

APK

Switch to advanced mode

PRODUCTION	BETA TESTING	ALPHA TESTING
Version 11009	Set up Beta testing for your app	Set up Alpha testing for your app

PRODUCTION CONFIGURATION — Upload new APK to Production

CURRENT APK published on **16 May 2014 03:06:02**

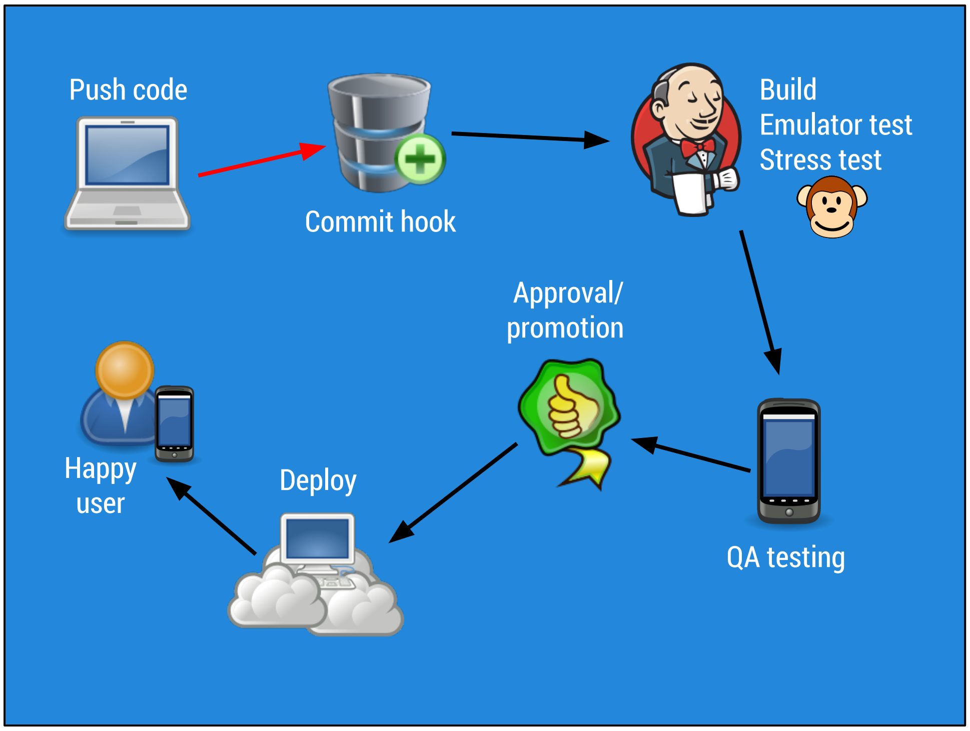
Supported devices	Excluded devices
3572	0
See list	Manage excluded devices

VERSION	UPLOADED ON	STATUS	ACTIONS
11009 (1.1)	16 May 2014	In Prod	

After all of this, we have a well-tested APK that we can upload to Google Play, or automatically upload to another testing or distribution service.
Here we see directly from the Google Play console that our app “In production” is version 1.1, and was built by Jenkins build #9.

Finally, while we’re looking at the Google Play upload interface...
Another thing you can automate with Jenkins, is to use matrix jobs and emulators to automatically create app store screenshots for multiple languages.

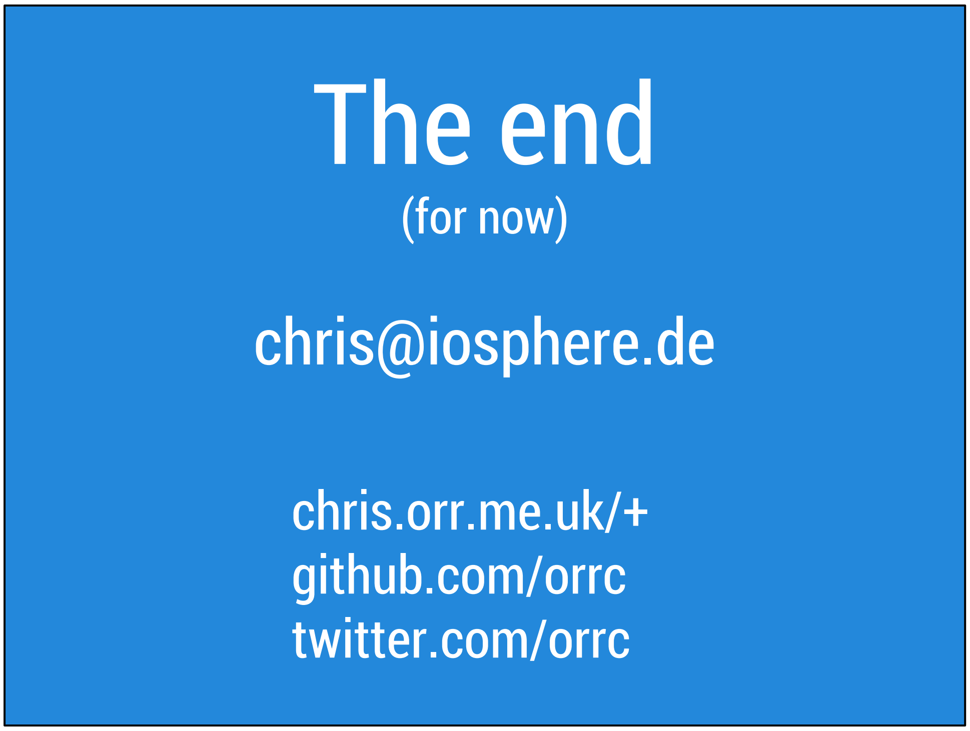
If you have five screenshots you want for each of five supported languages, setting up and taking 25 separate screenshots is pretty boring and error prone.
Instead, create a small test case with a library like Robotium, where you can set up various scenarios and take a screenshot for each one.
Then use a matrix job to run your tests multiple times, each time in a different emulator. A few minutes later, you will have all your required screenshots, archived by Jenkins and ready for upload.



Here’s an overview of the whole process we’ve discussed.

We can go from code pushed by a developer, with the app undergoing various types of testing — and optionally via manual QA testing and approval — finally being distributed to an end user.

So it’s possible that the **only** manual part required is the initial code push by the developer — everything else after this can be automated with Jenkins.


The end

(for now)

chris@iosphere.de

chris.orr.me.uk/+
github.com/orrc
twitter.com/orrc

Thanks a lot for listening / reading!

Let me know if you have any feedback.

Finally, if you're interested you can also check out the lightning talk I gave about Jenkins and the Git Plugin at JUC Berlin 2014.

Jenkins User Conference Europe

#jenkinsconf

Thank You To Our Sponsors

Platinum

CloudBees

Gold

zend The PHP Company

MidVision™ Release the innovation

codecentric

Silver

XebiaLabs the deployment automation company

SOASTA Test Faster. Release Sooner.

Corporate

cloudera®

Community

Java User Group Berlin Brandenburg

Lightweight Java User Group Munich