

res/ RRO OMS



Mårten Kongstad
@martenkongstad



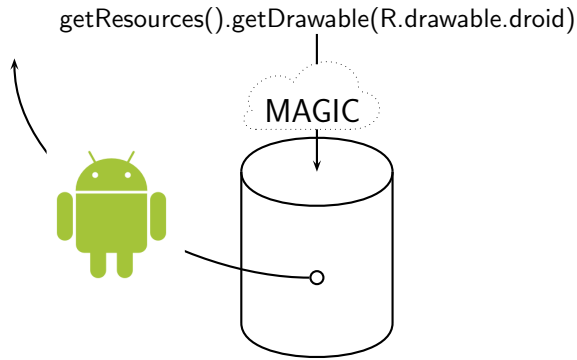
Zoran Jovanović
@jovzoran

- We are Mårten and Zoran, sw engineers from Sony Mobile Communications located in Lund, Sweden, and we've been working on Android since it became public.
- In this talk we shed some light on Android framework and application Resources - texts, images, that sort of stuff, an API that has been stable in Android at least since Android became public and then we talk about the evolution of its internals.
- If you ever fiddled with resources in R.java file, stared at it in attempt to understand the weird hex numbers in it OR had to rebuild the framework because resources got jumbled up somehow, this is where you might find some answers.

res/



- If you take away the byte code and the meta-data, what's left in your application are resources.
- Resources come in many different types: simple stuff like integers, more advanced types like strings, and complex things like drawables, layouts and xml.
- We've even seen games storing its levels as raw resources.



- On a high level, this is how you interact with resources. You call one of the get methods on a Resources object, some magic happens and the system returns a suitable value.
- We will see what's going on inside that magic cloud, but first we need to talk about some of the basic building block of resources.
- Let's start by having a look at what is inside an application APK.

```
$ unzip -l Demo.apk
```

```
Archive:  Demo.apk
```

Length	Date	Time	Name
-----	-----	-----	----
3568	2009-01-01	00:00	classes.dex
1712	2009-01-01	00:00	AndroidManifest.xml
611	2009-01-01	00:00	META-INF/CERT.SF
1722	2009-01-01	00:00	META-INF/CERT.RSA
528	2009-01-01	00:00	META-INF/MANIFEST.MF
1452	2009-01-01	00:00	resources.arsc
2864	2009-01-01	00:00	res/drawable/droid.xml
1112	2009-01-01	00:00	res/layout/activity_main.xml
-----			-----
13569			8 files

- An APK file is just a zip file.



```
$ unzip -l Demo.apk
```

```
Archive:  Demo.apk
```

Length	Date	Time	Name
-----	-----	-----	----
3568	2009-01-01	00:00	classes.dex
1712	2009-01-01	00:00	AndroidManifest.xml
611	2009-01-01	00:00	META-INF/CERT.SF
1722	2009-01-01	00:00	META-INF/CERT.RSA
528	2009-01-01	00:00	META-INF/MANIFEST.MF
1452	2009-01-01	00:00	resources.arsc
2864	2009-01-01	00:00	res/drawable/droid.xml
1112	2009-01-01	00:00	res/layout/activity_main.xml
-----			-----
13569			8 files

- There's the compiled byte code.



```
$ unzip -l Demo.apk
```

```
Archive:  Demo.apk
```

Length	Date	Time	Name
-----	-----	-----	----
3568	2009-01-01	00:00	classes.dex
1712	2009-01-01	00:00	AndroidManifest.xml
611	2009-01-01	00:00	META-INF/CERT.SF
1722	2009-01-01	00:00	META-INF/CERT.RSA
528	2009-01-01	00:00	META-INF/MANIFEST.MF
1452	2009-01-01	00:00	resources.arsc
2864	2009-01-01	00:00	res/drawable/droid.xml
1112	2009-01-01	00:00	res/layout/activity_main.xml
-----			-----
13569			8 files

- The manifest.



```
$ unzip -l Demo.apk
```

```
Archive:  Demo.apk
```

Length	Date	Time	Name
3568	2009-01-01	00:00	classes.dex
1712	2009-01-01	00:00	AndroidManifest.xml
611	2009-01-01	00:00	META-INF/CERT.SF
1722	2009-01-01	00:00	META-INF/CERT.RSA
528	2009-01-01	00:00	META-INF/MANIFEST.MF
1452	2009-01-01	00:00	resources.arsc
2864	2009-01-01	00:00	res/drawable/droid.xml
1112	2009-01-01	00:00	res/layout/activity_main.xml
13569			8 files

- Application meta-data.



```
$ unzip -l Demo.apk
```

```
Archive:  Demo.apk
```

Length	Date	Time	Name
3568	2009-01-01	00:00	classes.dex
1712	2009-01-01	00:00	AndroidManifest.xml
611	2009-01-01	00:00	META-INF/CERT.SF
1722	2009-01-01	00:00	META-INF/CERT.RSA
528	2009-01-01	00:00	META-INF/MANIFEST.MF
1452	2009-01-01	00:00	resources.arsc
2864	2009-01-01	00:00	res/drawable/droid.xml
1112	2009-01-01	00:00	res/layout/activity_main.xml
13569			8 files



- And finally, resources. The files under res/ shouldn't surprise you, but what is that resources.arsc file?


```
$ xxd resources.arsc | head
```

```
00000000: 0200 0c00 ac05 0000 0100 0000 0100 1c00 .....  
00000010: 9400 0000 0500 0000 0000 0000 0001 0000 .....  
00000020: 3000 0000 0000 0000 0000 0000 1900 0000 0.....  
00000030: 3800 0000 3f00 0000 5100 0000 1616 7265 8...?...Q....re  
00000040: 732f 6472 6177 6162 6c65 2f69 6d61 6765 s/drawable/droid  
00000050: 2e78 6d6c 001c 1c72 6573 2f6c 6179 6f75 .xml...res/layou  
00000060: 742f 6163 7469 7669 7479 5f6d 6169 6e2e t/activity_main.  
00000070: 786d 6c00 0404 4465 6d6f 000a 0f5b c390 xml...Demo...[..  
00000080: c3a9 e1b8 bfc3 b620 6f6e 655d 0008 10e2 ..... one]....  
00000090: 808f e280 ae44 656d 6fe2 80ac e280 8f00 .....Demo.....
```



- resources.arsc is the index of all resources in a package.
- resources.arsc is also pretty hard to pronounce so let's just refer to it as “the resource blob” from now on.
- The resource blob is a hash map on steroids, highly optimized for efficient lookup in runtime. It achieves this because:
 - It is a binary format.
 - It is created with the correct Endians-ness for most embedded devices.
 - And it is stored as an uncompressed file in the APK. This allows Android to simply memory map the blob without extracting it first.
- The actual grammar of the resource blob falls outside the scope of this talk.
- So Android uses the resource blob as an index. You may wonder how the blob is created. That is the job of ...

```
$ aapt package
  -c en_US,cs_CZ,da_DK,...,normal
  --preferred-density xxhdpi
  -M .../AndroidManifest.xml
  -S .../res
  -I .../framework-res.apk
  -F .../package.apk
```



- AAPT, the Android Asset Packaging Tool.
- The command shown here is the first step you take when building an application.
- AAPT will:
 - Convert your XML files (most notably the manifest) to a binary format for faster lookup on device, and
 - Parse your resource directory, creating:
 - ▶ The resource blob we just talked about, and
 - ▶ The R.java file.

```
$ aapt package
  -c en_US,cs_CZ,da_DK,...,normal
  --preferred-density xxhdpi
  -M .../AndroidManifest.xml
  -S .../res
  -I .../framework-res.apk
  -F .../package.apk
```



- Note how you specify the resource directory using the -S flag. AAPT doesn't care what the name of the resource directory is, but of course res/ is the de facto standard.

```
$ unzip -l package.apk
```

```
Archive: package.apk
```

Length	Date	Time	Name
1712	1980-01-01	01:00	AndroidManifest.xml
2864	1980-01-01	01:00	res/drawable/droid.xml
1112	1980-01-01	01:00	res/layout/activity_main.xml
1452	1980-01-01	01:00	resources.arsc
7140			4 files



- This is what your APK looks like after AAPT is done. It's still missing the byte code and the meta-data: later stages in the build chain will add those.
- AAPT does all this in one go. The downside to this is you waste time recalculating the same things over and over again when recompiling your application. The solution to that problem is ...

```
$ aapt2 compile -o ... ../res/layout/activity_main.xml
$ aapt2 compile -o ... ../res/mipmap-hdpi/ic_launcher.png
$ echo ../activity_main.xml.flat ../ic_launcher.png.flat >>
                                          ../aapt2-flat-list

$ aapt2 link
  -o ../package.apk
  --manifest ../AndroidManifest.xml
  -I ../framework-res.apk
  --java ../R.java
  --proguard ../proguard_options
  -c en_US,cs_CZ,da_DK,...,normal
  --preferred-density xxhdpi
  -R @../aapt2-flat-list [@overlay-flat-list-1
                        [@overlay-flat-list-2 [...]]]
```



- AAPT2. This tool creates the same artifacts as its predecessor, but it works more akin to a traditional build chain, with individual compilation steps and a final link step.
- AAPT2 is now default in Android Plugin for Gradle 3.0.

```
$ aapt dump --values resources app.apk
Package Groups (1)
Package Group 0 id=0x7f packageCount=1 name=com.sony.target
  Package 0 id=0x7f name=com.sony.target
    type 1 configCount=1 entryCount=1
      spec resource 0x7f020000 com.sony.target:drawable/droid: flags=...
      config (default):
        resource 0x7f020000 com.sony.target:drawable/droid: t=0x03 ...
        (string8) "res/drawable/droid.png"
    ...
```



- AAPT and AAPT2 can be used for things other than building.
- You can list the resources in a package...

```
$ aapt dump badging app.apk
package: name='com.sony.target' versionCode='1' versionName='1' ...
sdkVersion:'26'
targetSdkVersion:'26'
...
launchable-activity: name='com.sony.target.MainActivity' ...
feature-group: label=''
    uses-feature: name='android.hardware.faketouch'
    uses-implies-feature: name='android.hardware.faketouch' reason=...
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--_--' 'ar-XB' 'en-XA'
densities: '160'
```



- ...print meta-data ...

```

= ? V # ( . . _ _ _ _ a a , . | * ! S >
: . : ; = ; = Q Q Q Q B ' = : . . . :
. = ; ; ; : j Q Q Q Q f ; = ; ; = = : .
= ; ; ; : m Q Q Q Q Q = ; ; ; ; ; ; ; .
. = ; ; ; < Q Q Q Q W : ; ; ; ; ; = w ,
. / = ; ; : d Q Q Q Q # : ; ; ; ; : y Q [
. m : ; ; = Q Q Q Q Q m : ; ; ; ; = m Q Q L .
) W ; = ; ) Q Q Q Q Q Q ; = ; ; : m Q Q W ' .
< Q [ ; ; = Q Q Q Q Q W + ; ; ; < Q Q Q L
4 [ : - ; W Q Q Q Q F : : ; j Q Q Q [
] ( . ; d Q Q Q Q ( : . : d Q Q Q D
. . . . . - [ ; = m Q Q Q W . : = < Q Q P
. . = ; = = ; = ; ; ; : . . ) : : Q Q Q Q E : ; < Q Q D
: ; ; ; ; ; ; ; ; ; ; ; ; ; = . = ; ] Q Q Q Q k : ; m W D ' .

```




```

public final class R {
    ...
    public static final class string {
        public static final int app_name=0x7f050000;
        public static final int label=0x7f050001;
    }
    public static final class id {
        public static final int action_bar=0x7f060000;
        public static final int carrier=0x7f060001;
        public static final int edit_text=0x7f060002;
    }
}

```



- Let's move on to the other artefact of AAPT, the R.java file.
- For every resource type there is a corresponding inner class, and inside each class, every resource of the given type is enumerated with a unique numerical identifier.
- We'll spend some time on the actual numbers, but when coding, think of them as completely random. Any time you recompile your application, the values can potentially change.
- But wait a minute, you may say, if that's true, how come the SDK is backwards compatible? Why don't the numerical identifiers of the public resources in the SDK change between versions?
- Well, the truth is the framework cheats. It contains a file called public.xml which maps each public resource type and name to its numerical identifier. This file is used to lock resource identifiers in place.
- Resources internal to the framework still can and do change between versions, so as always, never rely on anything in the framework not explicitly public.

0x 7F 05 0001



- A resource numerical identifier is made up of three parts.

0x 7F 05 0001

package —┐



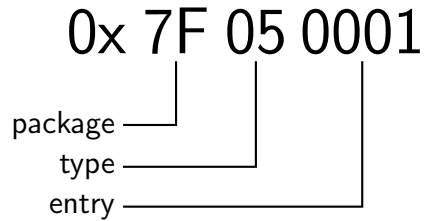
- The highest bits correspond to what type of package we're dealing with. 0x7f is by far the most commonly used value since it's used by applications, but there's also 0x01 used by the Android framework, and 0x00 for shared resource libraries.

0x 7F 05 0001

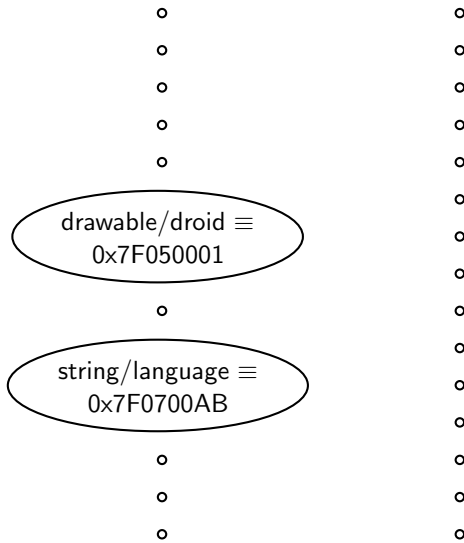
package ————┐
type —————┘



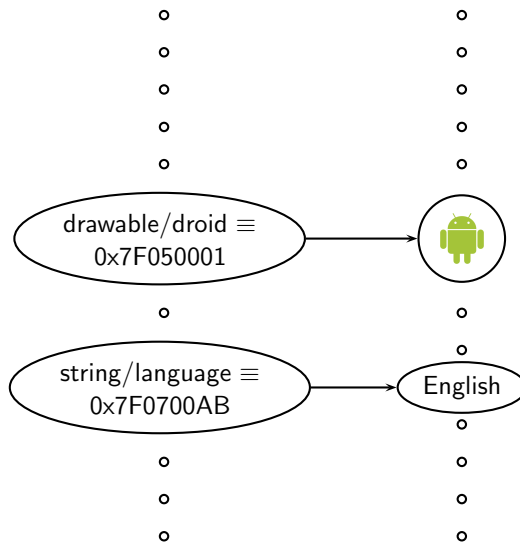
- The next few bits correspond to a resource type. There is no fixed mapping between types and values, so by just looking at this example we can't say if we're dealing with for instance a string or an integer, but within the same package, all resources of the same type will have the same type identifier.



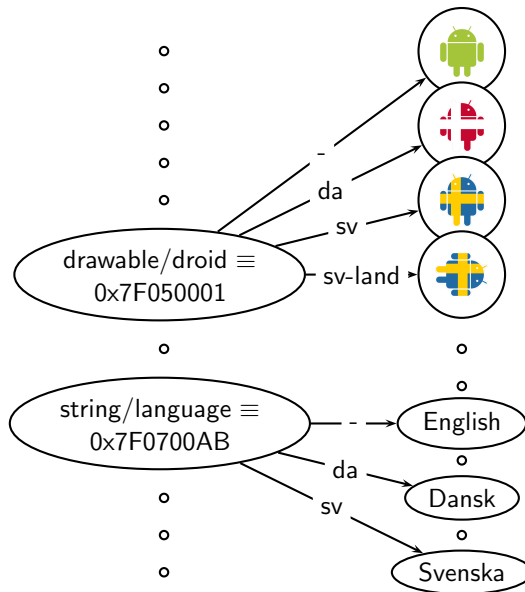
- Finally, there is the entry number. This is simply a running number.
- Reading the fields from right to left, in this example we're looking at the second resource of type 0x05 (whatever that is) in a regular app.



- So far we have focused on resources' types and names and how the R class maps these to numerical identifiers.
- Of course, during development we think of types and names, and in runtime Android works with numerical identifiers.



- Either way, the values are the keys in a dictionary.
- In the simplest case, resources are a 1-to-1 mapping.



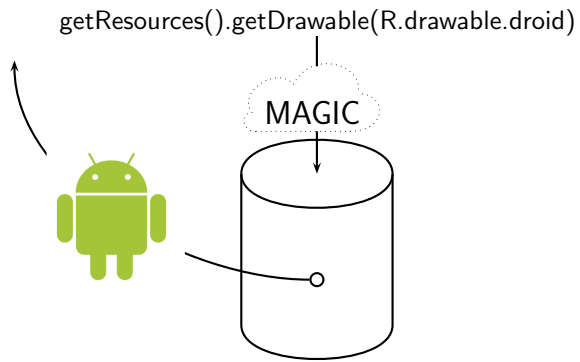
- But they can also be a 1-to-many mapping.
- And this is the real beauty of resources: the same key will map to different values depending on parameters outside the running application.
- You're all familiar with how you can provide translations for different languages or layouts for different orientations.
- You do so by tying values to parameters by placing resources in folders with special suffixes, like `values-sv` for Swedish locale.
- These parameters are called resource qualifiers. A set of qualifiers is called a configuration.
- Normally, you only tie one or two qualifiers to a resource but nothing prevents you from specifying all available qualifiers. And there are a lot of them.

```
$ adb exec-out cmd activity get-config | head -n1
```

```
config: 240mcc-1mnc-en-rUS-ldltr-sw411dp-w411dp-h659dp-normal-  
notlong-notround-lowdr-nowidecg-port-notnight-420dpi-finger-  
keysexposed-nokeys-navhidden-nonav-v27
```



- There are 22 qualifiers in this example, and more are added as Android continues to evolve!
- For example, Android Oreo added support for the wide color gamut qualifier.



- Remember this? We're finally ready to deep dive into the magic part!

```
getResources().getDrawable(R.drawable.droid)
```



- Again, first you call one of the get methods on a Resources object.

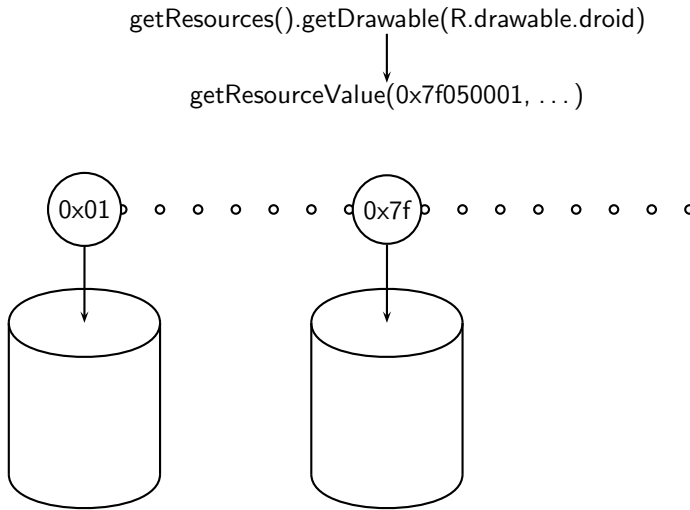
```
getResources().getDrawable(R.drawable.droid)
```



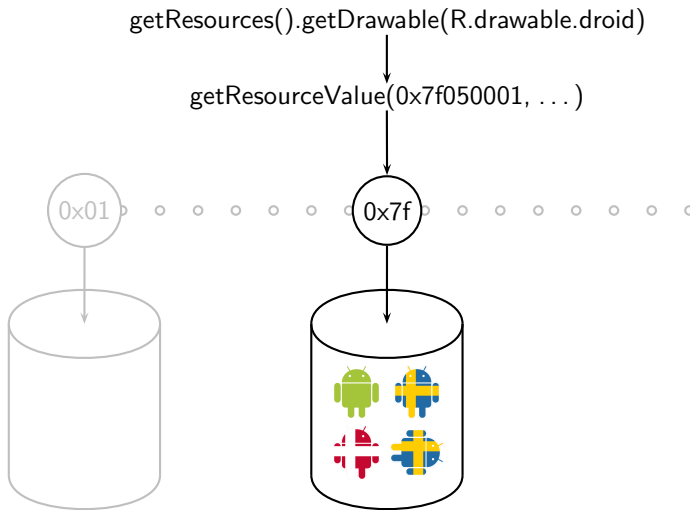
```
getResourceValue(0x7f050001, ...)
```



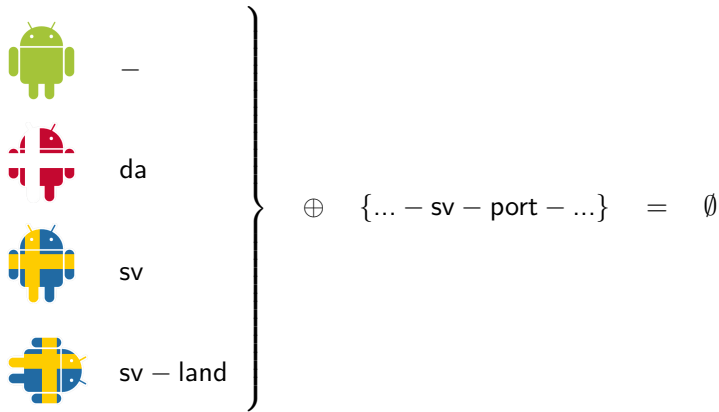
- All the get methods are eventually forwarded to the internal method `getResourceValue`. This is where you leave the Java world and enter native code, so every resource lookup comes with the cost of a JNI call.







- An application process loads not only its own APK, but also the framework APK, and potentially some others.
- At this point, the lookup code comes to a fork in the road: in which of the APKs should it continue to look for your resource?
- The answer lies in the first bits of the resource numerical identifier. In this example it's `0x7f`, so we continue with the application APK.



- In this example, the resource we're looking for is defined in four versions.

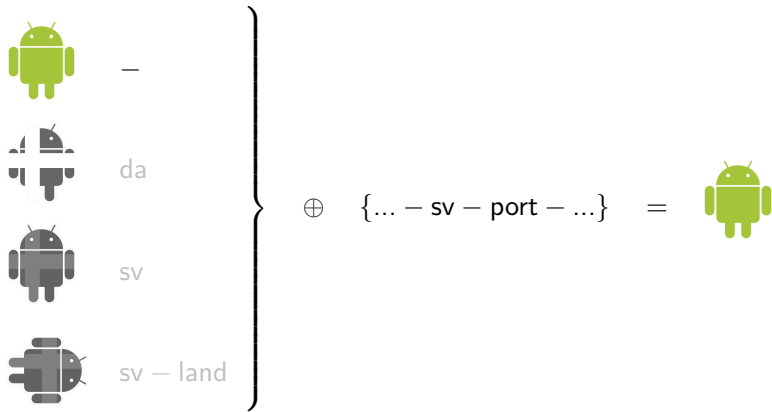


The diagram shows four Android icons with different color schemes and patterns, each followed by a label. These are grouped by a large right-facing curly brace. To the right of the brace is a mathematical expression involving a direct sum symbol (\oplus), a set of configurations, and an equals sign followed by an empty set symbol (\emptyset).

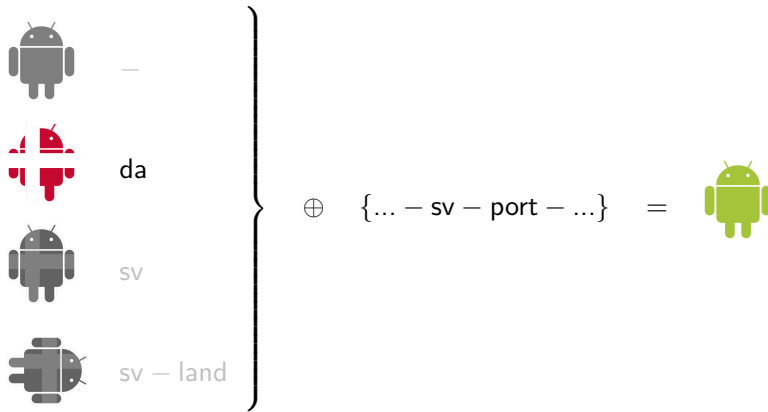
	—	}	$\oplus \quad \{ \dots - \text{sv} - \text{port} - \dots \} = \emptyset$
	da		
	sv		
	sv — land		



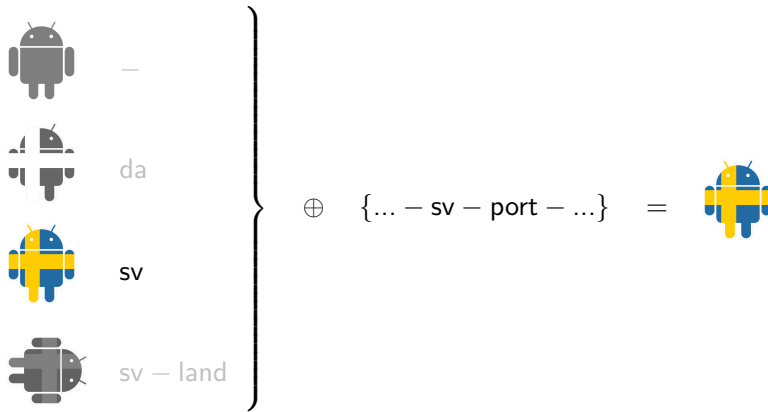
- The framework will return what it considers “the best match” given a requested configuration. Most of the time, the requested configuration corresponds to the device’s current configuration. In this example, the requested configuration is something-something-Swedish-portrait.
- The framework will consider every version of a resource while keeping track of the best match found so far.
- For each version, there is a check to see if it is compatible with the requested configuration, and if it is, there is another check to see if it is better than the current best match.



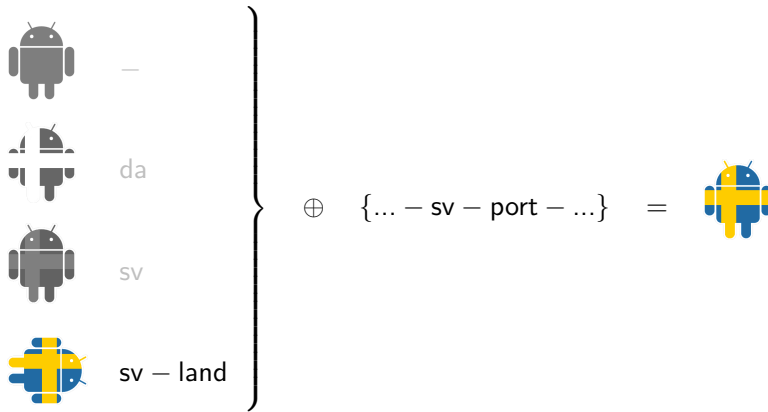
- Let's start with the default version.
- By definition default is compatible with every configuration, and that's why the system will always fall back to default if nothing else matches.
- For now, this version is the best match.



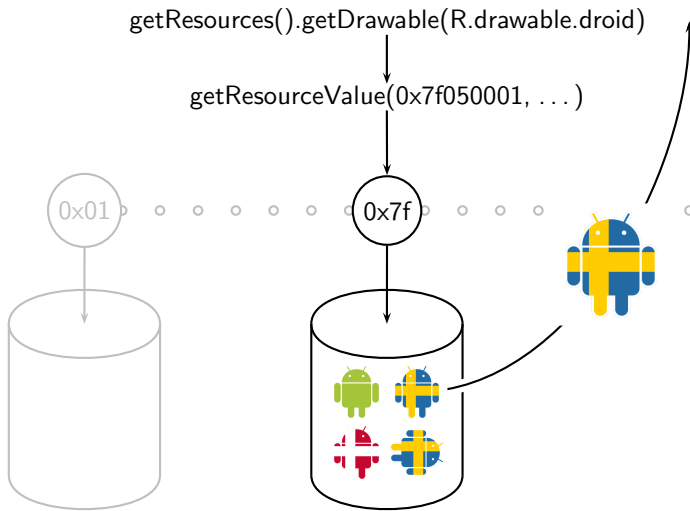
- Next up is Danish. Android doesn't consider Danish and Swedish compatible, so this version is skipped.



- Next in line is Swedish. This is compatible with the requested configuration, and is considered a better match, so Swedish takes over as the current best match.
- You have to look at the source code to see exactly what makes one version a better match than another, but as a rule of thumb, the more qualifiers, the better.



- Finally, we come to Swedish Landscape, but because Landscape isn't compatible with Portrait, this version is skipped.
- The best match turned out to be Swedish ...



- And that's the value that is returned to the application.

```

struct Res_value
{
    typedef uint32_t data_type;
    enum : uint8_t {
        TYPE_NULL = 0x00,
        TYPE_REFERENCE = 0x01,
        TYPE_STRING = 0x03,
        TYPE_FLOAT = 0x04,
        ...
    };

    uint16_t size;
    uint8_t res0; // Padding, always 0
    uint8_t dataType;
    data_type data;
};

```



- In the resource blob, resource values are encoded using this structure. The dataType says what type of resource it is, and the data field is interpreted accordingly.

```

struct Res_value
{
    typedef uint32_t data_type;
    enum : uint8_t {
        TYPE_NULL = 0x00,
        TYPE_REFERENCE = 0x01,
        TYPE_STRING = 0x03,
        TYPE_FLOAT = 0x04,
        ...
    };

    uint16_t size;
    uint8_t res0; // Padding, always 0
    uint8_t dataType; // TYPE_INT_BOOLEAN, TYPE_INT_DEC, ...
    data_type data;
};

```



- If the resource is a simple type, like a boolean or an integer, its value can be fully represented by the four bytes in the data field.

```

struct Res_value
{
    typedef uint32_t data_type;
    enum : uint8_t {
        TYPE_NULL = 0x00,
        TYPE_REFERENCE = 0x01,
        TYPE_STRING = 0x03,
        TYPE_FLOAT = 0x04,
        ...
    };

    uint16_t size;
    uint8_t res0; // Padding, always 0
    uint8_t dataType; // TYPE_STRING
    data_type data;
};

```



- If it's a string, the data field is an integer offset into another part of the resource blob where string are stored, so the framework will do an additional lookup before returning your string.


```

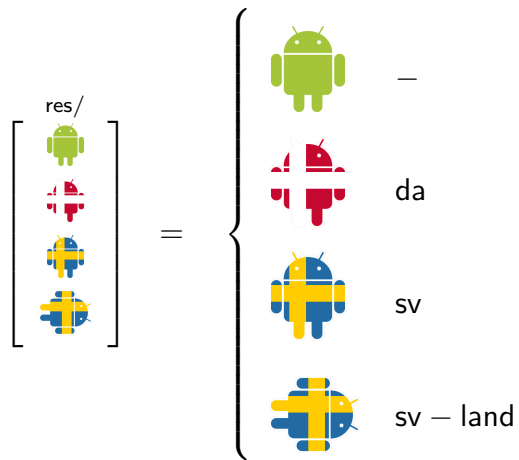
struct Res_value
{
    typedef uint32_t data_type;
    enum : uint8_t {
        TYPE_NULL = 0x00,
        TYPE_REFERENCE = 0x01,
        TYPE_STRING = 0x03,
        TYPE_FLOAT = 0x04,
        ...
    };

    uint16_t size;
    uint8_t res0; // Padding, always 0
    uint8_t dataType; // TYPE_STRING -> .xml, drawable/, ...
    data_type data;
};

```

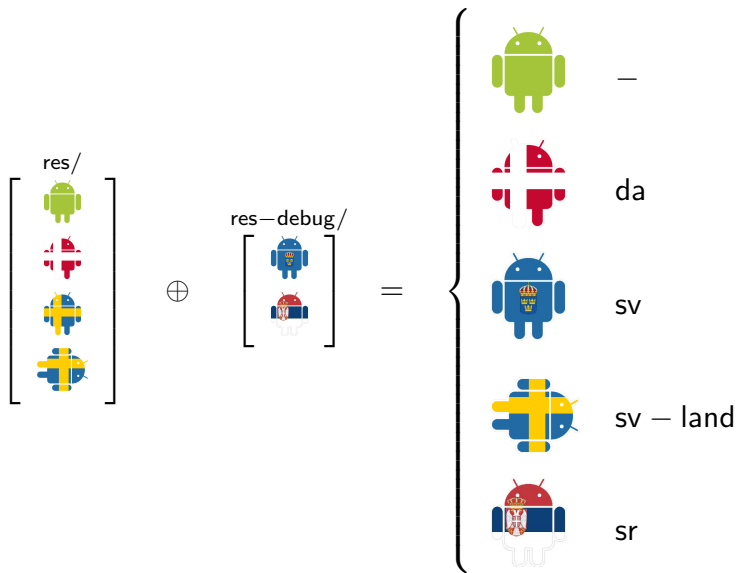


- Finally, if the resource type is a drawable or an XML, its contents are stored outside the resource blob as a separate file in the APK zip. In this case, the data value is again an offset to a string, and the string's value is the contents' path within the APK.



- Remember how we told AAPT to look for resources using the -S flag?



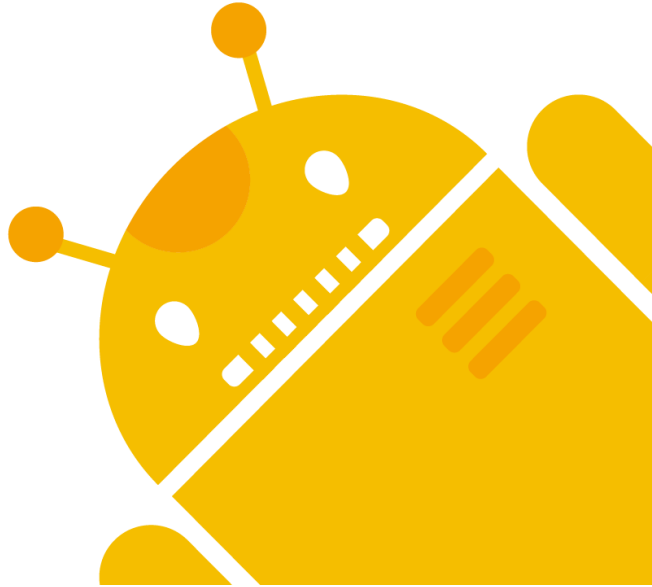


- You can actually pass in several `-S` flags, in which case some directories will act like diffs, patching resources in others.
- This is called static overlay, and it can be useful if you want to add different resource values for your debug build, or if you're a modder and want to modify core apps without touching their code.
- In some cases static overlay is the right choice, but the downside to is that everything happens in build time, and sometimes that simply isn't good enough.



- There are several modularization techniques in Android, e.g.:
 - Splits - which allow you to build different versions of APKs depending on supported density or abi
 - Android Archives - JARs packaged with additional Android info
- but this is not a talk about those. They deserve talks of their own. Instead we focus on something somewhat similar.

RRO



- As device manufacturers we had to customize a value in the framework and compile time customization was prohibitively expensive (the change needed to affect many but not all customers and we needed to optimize time spent in the factory). There was a bug in our implementation and we found the solution to be applicable to many more use cases.
- It eventually evolved into something we now call Runtime Resource Overlay and has since 2010 been a part of AOSP in several iterations.
- So, what is Runtime Resource Overlay?

a way to OVERLAY the RESOURCES of an app in RUNTIME



- It's a way to OVERLAY, that is extend or modify the RESOURCES of an application or framework in RUNTIME - i.e. post build on a running device and without reboot, without interruptions that could cause ANRs or janks and without affecting the code of the app.
- When is this useful?

Customization



- Software customization - the bread and butter of device manufacturers - is among other things a configurable software deployed using one build but multiple configurations to modify behavior of software in runtime, so a runtime resource overlay will typically be a configuration for a specific carrier, network, region. . .
- e.g. to configure the device with specific network parameters or to comply to a regional law etc.

Themes



- Personalized themes change look and feel of software, by changing some color values or graphical artefacts which are usually defined as resources. . .



Transparency



Performance



Security



Memory



- Having the origin of the idea and these use cases in mind we set some design goals:
- the process should be **TRANSPARENT**, to the system, to a developer and to end user,
- There will be no changes in the existing APIs, so all apps just continue working as they were and development process doesn't change for developers, you don't have to use this if you don't want to and the entry bar should be really low if you do want to use it.
- For an end user, there will be no reboots or interruptions that could cause ANRs or janks.
- So, everything has to be really **FAST**. Load and lookup will have no noticeable difference from load and lookup of regular resources. Remember how resource blob file is mmaped? We wanted direct access to new resources as well as the original ones,
- which is why everything has to remain **SECURE** and we needed to be able to reuse Android's app security model with permissions and signatures.
- On top of all that, since we are adding new resources, it's important to have **LOW MEMORY FOOTPRINT**. So no databases, no big markup files, like xml etc. It appealed to us how resource blob is compact, dense and terse.

```
$ unzip -l DemoOverlay.apk
Archive:  DemoOverlay.apk
  Length      Date    Time    Name
-----
 146035  2009-01-01  00:00    res/drawable/droid.jpg
    588  2009-01-01  00:00    resources.arsc
   1068  2009-01-01  00:00    AndroidManifest.xml
    428  2009-01-01  00:00    META-INF/CERT.SF
   1722  2009-01-01  00:00    META-INF/CERT.RSA
    345  2009-01-01  00:00    META-INF/MANIFEST.MF
-----
 150186                                6 files
```



- That's why early on we decided that the physical representation of Runtime Resource Overlay should be an APK - a regular package that PackageManager knows about, so all the regular rules of package management apply.
- You would build this package using whatever build chain you use today.
- See how if you unzip the overlay package it will contain ***ALL*** the building blocks of an APK ***EXCEPT*** the code.

```
$ unzip -l DemoOverlay.apk
Archive:  DemoOverlay.apk
  Length      Date    Time    Name
-----
 146035  2009-01-01  00:00    res/drawable/droid.jpg
    588  2009-01-01  00:00    resources.arsc
   1068  2009-01-01  00:00    AndroidManifest.xml
    428  2009-01-01  00:00    META-INF/CERT.SF
   1722  2009-01-01  00:00    META-INF/CERT.RSA
    345  2009-01-01  00:00    META-INF/MANIFEST.MF
-----
 150186
                        6 files
```

- There's a manifest...



```
$ unzip -l DemoOverlay.apk
Archive:  DemoOverlay.apk
  Length      Date    Time    Name
-----
 146035  2009-01-01  00:00    res/drawable/droid.jpg
    588  2009-01-01  00:00    resources.arsc
   1068  2009-01-01  00:00    AndroidManifest.xml
    428  2009-01-01  00:00    META-INF/CERT.SF
   1722  2009-01-01  00:00    META-INF/CERT.RSA
    345  2009-01-01  00:00    META-INF/MANIFEST.MF
-----
 150186
                        6 files
```



- There's metadata - including signature. . .

```
$ unzip -l DemoOverlay.apk
Archive:  DemoOverlay.apk
  Length      Date    Time    Name
-----
 146035  2009-01-01  00:00    res/drawable/droid.jpg
    588  2009-01-01  00:00    resources.arsc
   1068  2009-01-01  00:00    AndroidManifest.xml
    428  2009-01-01  00:00    META-INF/CERT.SF
   1722  2009-01-01  00:00    META-INF/CERT.RSA
    345  2009-01-01  00:00    META-INF/MANIFEST.MF
-----
 150186                                6 files
```



- ***AND*** there are resources.
- These are specifically the resources selected from and defined in the original, let's call it target, package whose resources we want to modify, but no other resource from the target package should be in the overlay. (It's like a sort of a diff.)
- Resources are picked by the type and name, the fully qualified name of a resource if you will - almost as it's sort of an API.
- Note again how there is no code in this package. Not even R class with definitions of numerical identifiers.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.sony.overlay"
  android:versionCode="2"
  android:versionName="2.0" >
  <uses-sdk
    android:minSdkVersion="26"
    android:targetSdkVersion="26" />

    <overlay android:targetPackage="com.sony.target" />
</manifest>
```



- The manifest of a resource overlay looks like an ordinary manifest file everybody's used to

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.sony.overlay"
  android:versionCode="2"
  android:versionName="2.0" >
  <uses-sdk
    android:minSdkVersion="26"
    android:targetSdkVersion="26" />

    <overlay android:targetPackage="com.sony.target" />
</manifest>
```



- There's a package name.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.sony.overlay"
  android:versionCode="2"
  android:versionName="2.0" >
  <uses-sdk
    android:minSdkVersion="26"
    android:targetSdkVersion="26" />

    <overlay android:targetPackage="com.sony.target" />
</manifest>
```



- There's version info etc.


```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.sony.overlay"
  android:versionCode="2"
  android:versionName="2.0" >
  <uses-sdk
    android:minSdkVersion="26"
    android:targetSdkVersion="26" />

    <overlay android:targetPackage="com.sony.target" />
</manifest>
```



- But the most important element you wouldn't find in the manifest of an app is this 'overlay' element which identifies the runtime resource overlay package.
- The value of its attribute 'targetPackage' is the package name of the package, framework or application, which defines the resources that you want to modify.
- If you're into git archeology and read the code of 2nd iteration commits in AOSP, you will notice that there used to be a 'priority' attribute which described the wanted order in which to apply the overlay package, but we will soon show why that attribute was deprecated.
- So, to build the runtime resource overlay package, all you need is this manifest to point to the target package and then some resources.

```
getResources().getDrawable(R.drawable.droid)
```

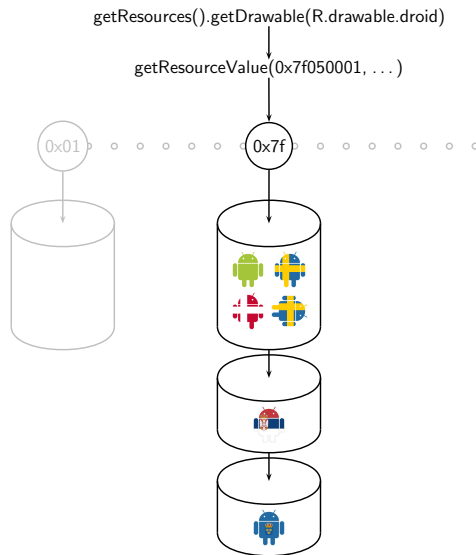


- Lookup has not changed.
- In code, you'd ask for a resource in the same way as you did before, no changes to the code whatsoever. You would use the type and name of the resource defined in R class of the target package.

```
getResources().getDrawable(R.drawable.droid)  
    ↓  
getResourceValue(0x7f050001, ...)
```

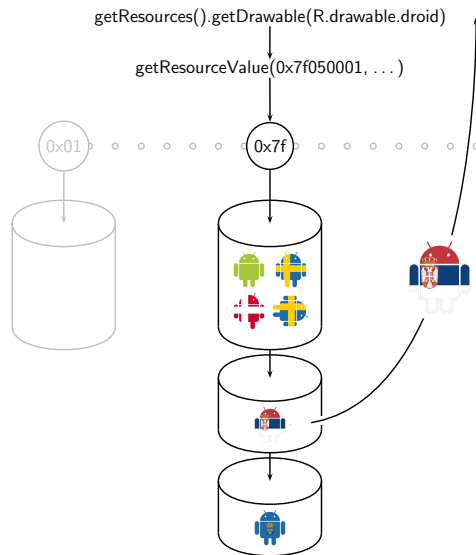


- That would invoke `getResourceValue` with the actual value of the numerical identifier defined in the target package.



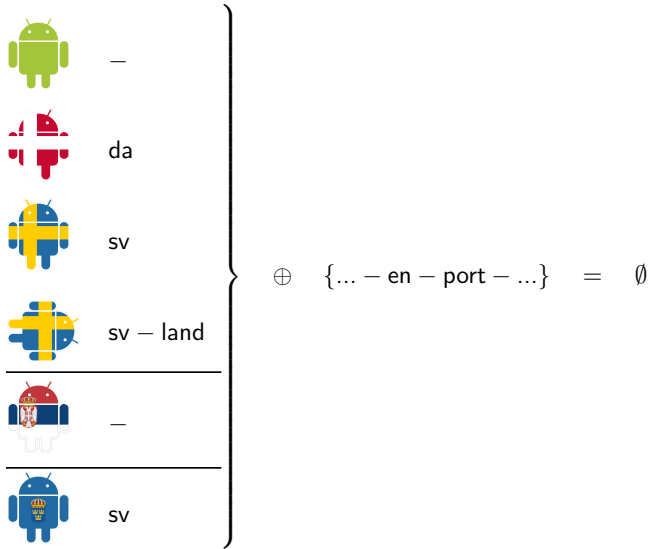
- ... System will look for resources in the correct package - “7f for the application” ...



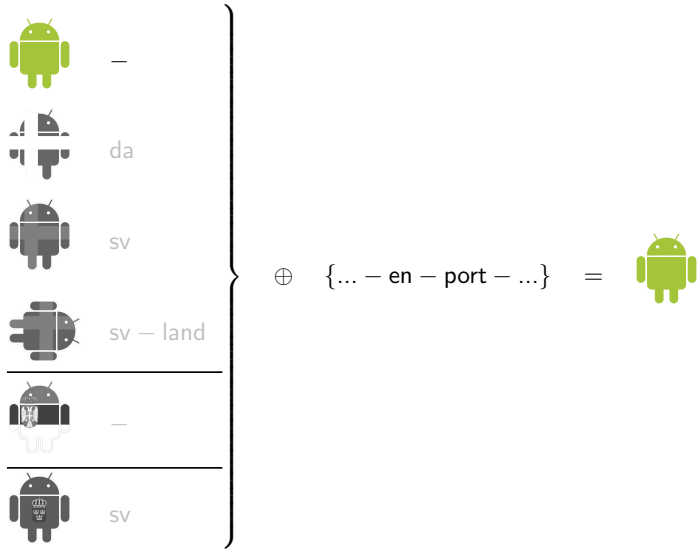


- ...and eventually the best match is returned.

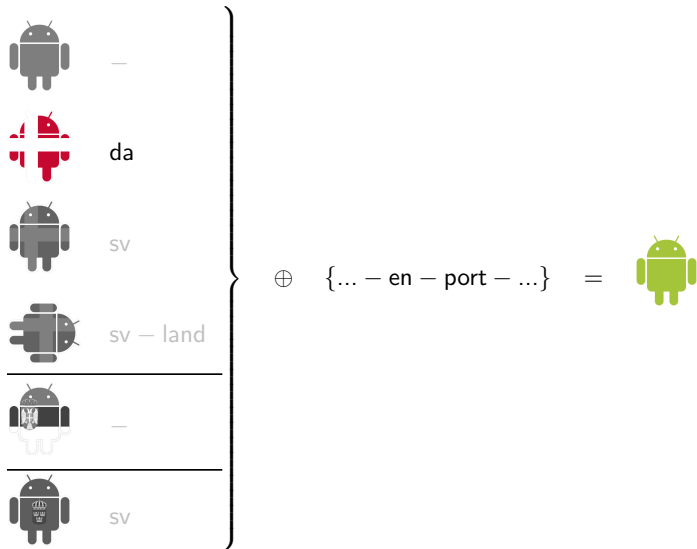




- Let's zoom in, e.g. we have a device in English portrait configuration and we are looking for a drawable droid.
- In this example, again we have 4 versions of drawable droid defined in the app, and additionally we define 2 versions in a separate overlay package each.

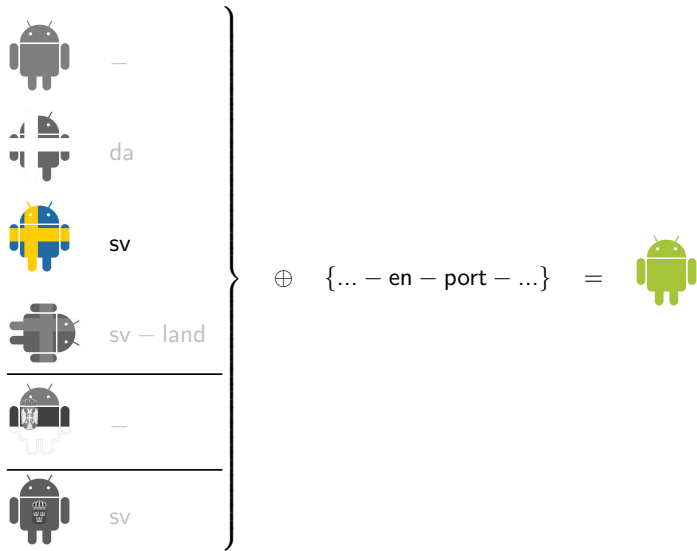


- Default resource fits any qualifier.

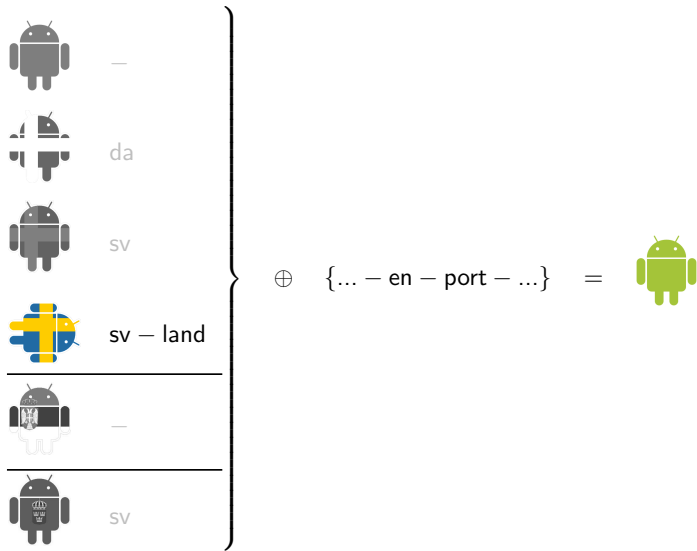


- And it's a better match than Danish...



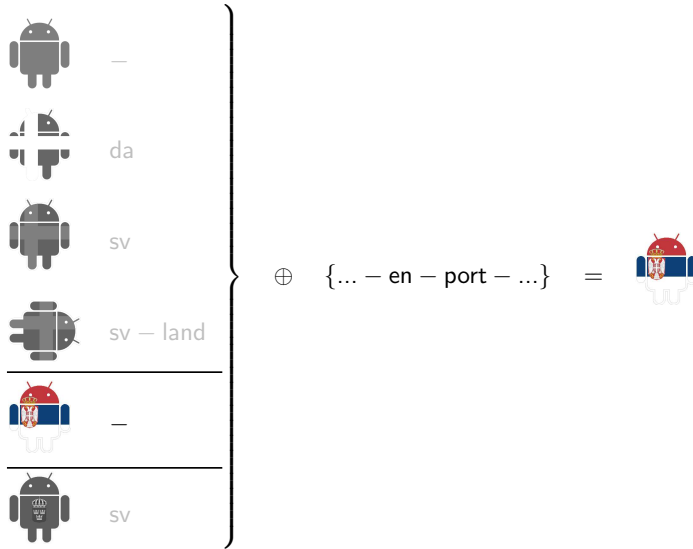


- ...or Swedish ...



- ...or Swedish landscape ...





- But the default resource defined in overlay package takes over as the best match.
- With the same qualifiers - overlay resource is a better match than the resource in the target package.
- Notice one pitfall here. Remember "the more qualifiers the better"? Well, with the right qualifiers, resources from the target apk can still be a better match, but in this example the overlay replaces the default. So if you had a Danish locale selected, then that version of resource defined in the target package would win as the best match.
- Now, during lookup, framework doesn't know exactly which apk will contain the best match, so it will go through all 7f packages.
- Is performance an issue? System does look in each apk, but the lookup is really fast, especially when skipping packages that do not specify the requested entry and then the resource blob is designed for quick jumps to resources thanks to the numerical identifiers.
- But how does the system know where to look if overlay packages don't have numerical identifiers in them and you don't need them in order to build the overlay package?

```
$ adb exec-out idmap
```



- We developed a tool that runs on device - idmap tool, whose purpose is to create a 1 to 1 mapping between overlay and its target, i.e. between fully qualified resource names in overlay package and corresponding numerical resource identifiers in the target package.
- One neat thing with the resource blobs of framework and apps is that they contain all the information about the resources - the name, the type and the numerical identifier of the resource. Idmap will do a reverse lookup from type and name, fetch the numerical identifier and make a record of it and store it in a binary file.

```
$ adb exec-out idmap --inspect /data/resource-cache/...base.apk@idmap
SECTION      ENTRY      VALUE      COMMENT
IDMAP HEADER magic      0x504d4449
              version    0x00000002
              base crc   0x414dc2c3
              overlay crc 0x31702a90
              base path   ..... /data/app/.../base.apk
              overlay path ..... /vendor/overlay/.../base.apk
DATA HEADER  target pkg  0x0000007f
              types count 0x00000001
DATA BLOCK   target type 0x00000002
              overlay type 0x00000002
              entry count  0x00000001
              entry offset 0x00000000
              entry       0x00000000 drawable/droid
```



- This is a pretty-print version of a sample idmap file.
- Its binary format is optimized for fast lookup.

```
$ adb exec-out idmap --inspect /data/resource-cache/...base.apk@idmap
```

SECTION	ENTRY	VALUE	COMMENT
IDMAP HEADER	magic	0x504d4449	
	version	0x00000002	
	base crc	0x414dc2c3	
	overlay crc	0x31702a90	
	base path	/data/app/.../base.apk
	overlay path	/vendor/overlay/.../base.apk
DATA HEADER	target pkg	0x0000007f	
	types count	0x00000001	
DATA BLOCK	target type	0x00000002	
	overlay type	0x00000002	
	entry count	0x00000001	
	entry offset	0x00000000	
	entry	0x00000000	drawable/droid



- You can issue commands like this in adb to use idmap. Here we inspect, i.e. pretty-print or dump an idmap file.

```
$ adb exec-out idmap --inspect /data/resource-cache/...base.apk@idmap
```

SECTION	ENTRY	VALUE	COMMENT
IDMAP HEADER	magic	0x504d4449	
	version	0x00000002	
	base crc	0x414dc2c3	
	overlay crc	0x31702a90	
	base path	/data/app/.../base.apk
	overlay path	/vendor/overlay/.../base.apk
DATA HEADER	target pkg	0x0000007f	
	types count	0x00000001	
DATA BLOCK	target type	0x00000002	
	overlay type	0x00000002	
	entry count	0x00000001	
	entry offset	0x00000000	
	entry	0x00000000	drawable/droid



- There's an idmap header that tells you where the idmap begins, some integrity protection info AND the path to the target package...


```
$ adb exec-out idmap --inspect /data/resource-cache/...base.apk@idmap
```

SECTION	ENTRY	VALUE	COMMENT
IDMAP HEADER	magic	0x504d4449	
	version	0x00000002	
	base crc	0x414dc2c3	
	overlay crc	0x31702a90	
	base path	/data/app/.../base.apk
	overlay path	/vendor/overlay/.../base.apk
DATA HEADER	target pkg	0x0000007f	
	types count	0x00000001	
DATA BLOCK	target type	0x00000002	
	overlay type	0x00000002	
	entry count	0x00000001	
	entry offset	0x00000000	
	entry	0x00000000	drawable/droid



- There's a data header with the target package identifier, in this case 7f for app, and there's also a count of types that this file maps to.

```
$ adb exec-out idmap --inspect /data/resource-cache/...base.apk@idmap
```

SECTION	ENTRY	VALUE	COMMENT
IDMAP HEADER	magic	0x504d4449	
	version	0x00000002	
	base crc	0x414dc2c3	
	overlay crc	0x31702a90	
	base path	/data/app/.../base.apk
	overlay path	/vendor/overlay/.../base.apk
DATA HEADER	target pkg	0x0000007f	
	types count	0x00000001	
DATA BLOCK	target type	0x00000002	
	overlay type	0x00000002	
	entry count	0x00000001	
	entry offset	0x00000000	
	entry	0x00000000	drawable/droid

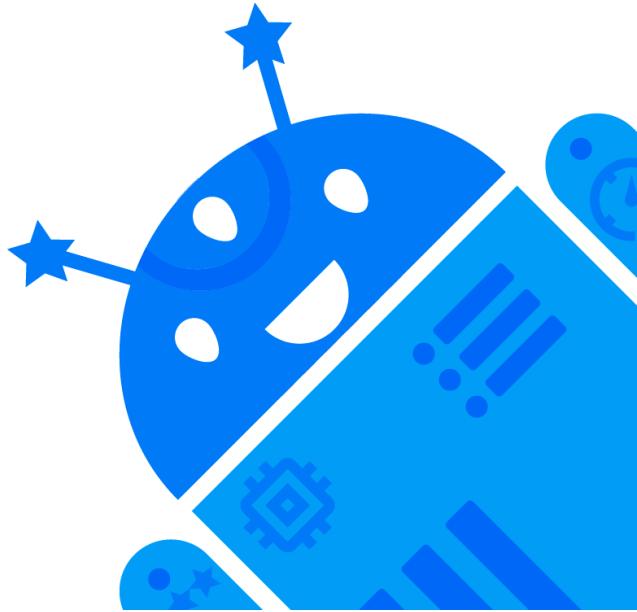


- And then there's the data itself. The mapping to the individual resource that's modified. The package, the type and entry - it's all there.

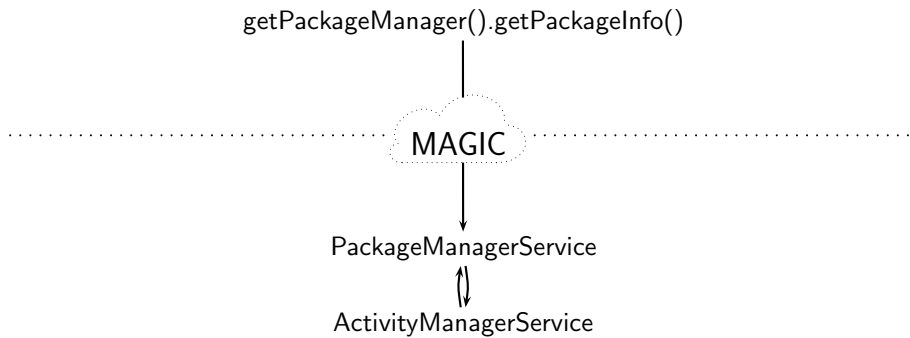


- RRO Packages:
 - Have to be stored in the correct path on device,
 - Can only be preloaded by device manufacturers / platform builders,
 - and until Oreo they were loaded automatically just by virtue of having the apk in the correct path
- But this wasn't enough.

OMS



- The use cases we mentioned earlier, customization and theming, really required a more dynamic access to overlays, something that allows you to rearrange overlays on-the-fly.
- We solved this by adding a new system component that we chose to call the overlay manager service, OMS for short.
- By the way, all the old design goals are still valid. You remember, performance, security and all that.



- A quick side note about the internal structure of the Android framework. It consists of several modules called *system services*.
- The wording is a bit confusing: system services are nothing like the service class used by applications.
- They don't have a lifecycle, are always running, and are started as part of the Android boot in a hard-coded order.
- System services often contain both internal methods for the framework as well as public facing methods for applications.
- When you access a manager in your application, it's usually backed up by a corresponding system service.
- For instance, calls made to the package manager are forwarded to the package manager service.
- This link between the application and the framework is implemented as an AIDL interface.

```
interface IOverlayManager {
    Map getAllOverlays(in int userId);
    List getOverlayInfosForTarget(in String targetPackageName,
                                   in int userId);
    OverlayInfo getOverlayInfo(in String packageName, in int userId);
    boolean setEnabled(in String packageName, in boolean enable,
                       in int userId);
    boolean setEnabledExclusive(in String packageName,
                                in boolean enable, in int userId);
    boolean setPriority(in String packageName,
                        in String newParentPackageName, in int userId);
    boolean setHighestPriority(in String packageName, in int userId);
    boolean setLowestPriority(in String packageName, in int userId);
}
```



- This is the AIDL interface for the overlay manager service.
- If you're not familiar with the AIDL language, just squint your eyes and think of it as regular Java.

```
interface IOverlayManager {
    Map getAllOverlays(in int userId);
    List getOverlayInfosForTarget(in String targetPackageName,
                                   in int userId);
    OverlayInfo getOverlayInfo(in String packageName, in int userId);
    boolean setEnabled(in String packageName, in boolean enable,
                       in int userId);
    boolean setEnabledExclusive(in String packageName,
                                in boolean enable, in int userId);
    boolean setPriority(in String packageName,
                        in String newParentPackageName, in int userId);
    boolean setHighestPriority(in String packageName, in int userId);
    boolean setLowestPriority(in String packageName, in int userId);
}
```



- There are methods to tell the overlay manager to change something about the overlays.
- The setPriority methods change the order in which overlays are loaded. This is what deprecated the manifest attribute.
- The setEnabled methods toggle if an overlay should be used in the first place.
- These methods will trigger affected applications to reload their resources.

```
interface IOverlayManager {
    Map getAllOverlays(in int userId);
    List getOverlayInfosForTarget(in String targetPackageName,
                                   in int userId);
    OverlayInfo getOverlayInfo(in String packageName, in int userId);
    boolean setEnabled(in String packageName, in boolean enable,
                       in int userId);
    boolean setEnabledExclusive(in String packageName,
                                in boolean enable, in int userId);
    boolean setPriority(in String packageName,
                       in String newParentPackageName, in int userId);
    boolean setHighestPriority(in String packageName, in int userId);
    boolean setLowestPriority(in String packageName, in int userId);
}
```



- There are also methods to get the current status of overlays.
- They all return a new class OverlayInfo, or at least a collection containing OverlayInfos.


```
public final class OverlayInfo implements Parcelable {  
    public static final int STATE_UNKNOWN = -1;  
    public static final int STATE_MISSING_TARGET = 0;  
    public static final int STATE_NO_IDMAP = 1;  
    public static final int STATE_DISABLED = 2;  
    public static final int STATE_ENABLED = 3;  
  
    public final int state;  
    public final String packageName;  
    public final String targetPackageName;  
    public final String baseCodePath;  
    public final int userId;  
}
```



- Because overlays are regular packages, you can still ask the package manager about their general information, but overlay details are stored in OverlayInfos.

```
public final class OverlayInfo implements Parcelable {  
    public static final int STATE_UNKNOWN = -1;  
    public static final int STATE_MISSING_TARGET = 0;  
    public static final int STATE_NO_IDMAP = 1;  
    public static final int STATE_DISABLED = 2;  
    public static final int STATE_ENABLED = 3;  
  
    public final int state;  
    public final String packageName;  
    public final String targetPackageName;  
    public final String baseCodePath;  
    public final int userId;  
}
```



- The most important part is the overlay's state.
- An overlay can be
 - *enabled*, meaning it's loaded, or
 - *disabled*, meaning it's not loaded, but can be, or
 - in one of the error states that prevent it from being loaded: for instance, maybe its target isn't installed.



- Runtime resource overlay relies on the security inherent in the resource framework.
- But adding a Java interface brought in a new attack vector.
- Without these security measures in place, a malicious overlay could change the customization, crash your device or swap the yes and no texts in a dialog box.
- Hardening AIDL interfaces is usually done via Android permissions and signature checks.
- For the overlay manager, we reused an existing permission for the read operations and introduced a new permission for the write operations.

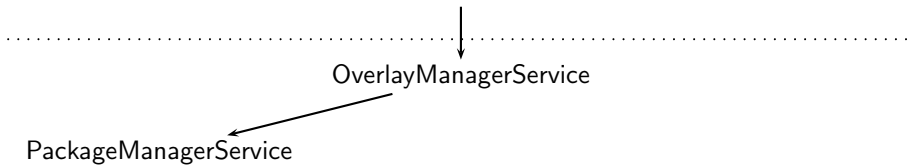


- In the talk we show how interaction with the overlay manager looks like on an actual device.
- In the video, we're using a Sony Xperia X running AOSP thanks to the Open Devices program.

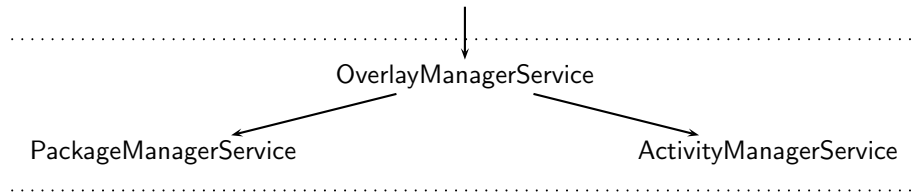
.....
↓
OverlayManagerService



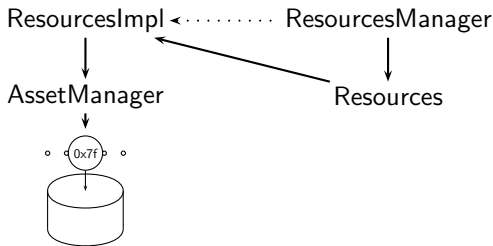
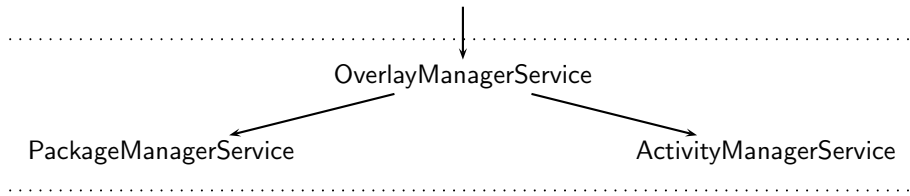
- So what happened in the video? Let's examine what the system does when an overlay is enabled.
- First, a client tells the overlay manager to change something. In the video we did this via the adb shell, but it could be anything with access to AIDL, like a theme picker application.
- The dotted line indicates that we're moving from one process to another.



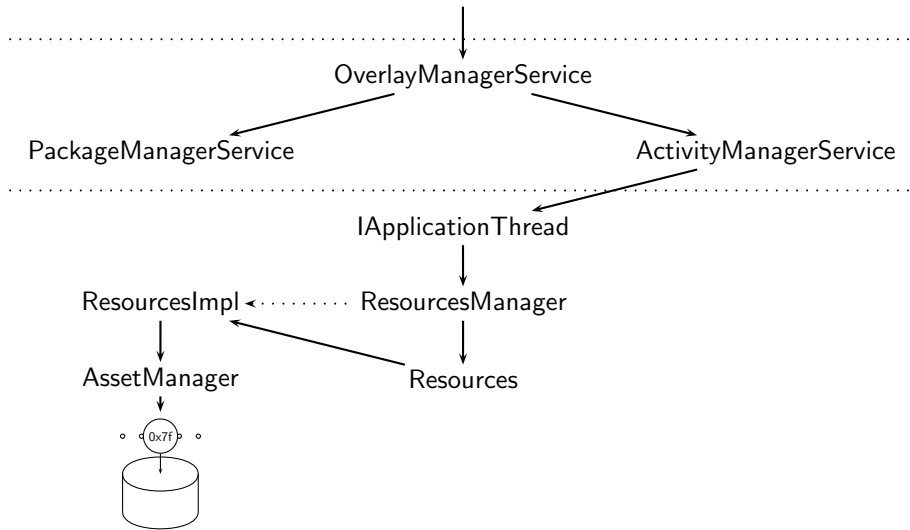
- If the request is accepted, the overlay manager tells the package manager about the change in APK paths. Any application started after this point will be correctly set up.



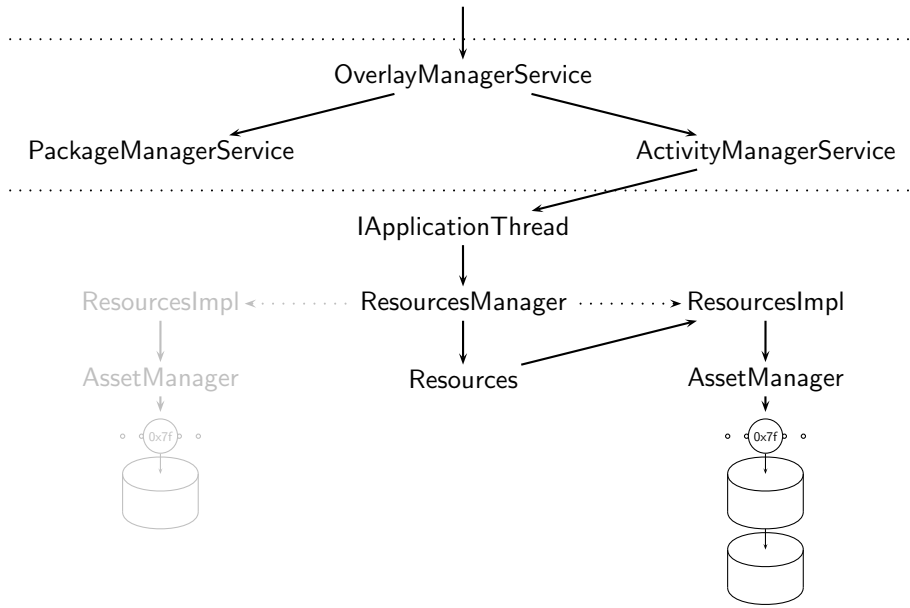
- But we also need to handle any running processes, so next, the overlay manager asks the activity manager for help.
- The activity manager will go through every running process, and for each application affected by the overlay change, send that process a message.



- Before we continue further, let's see how processes load resources.
- Your application interacts with the Resources class, but Resources simply forward your calls to resource implementations which are what control your resource blobs.
- Everything is orchestrated by the resources manager.
- Because your application only knows about Resources, the manager can swap out the underlying implementation at any time.



- So back to the message from the activity manager. It's picked up on your application's main thread.



- This triggers the resources manager to create a new resources implementation and to redirect the Resources object.
- Your application performs a configuration change, and the new resources are in use.
- And that, in a nutshell, is how the overlay manager service works.

res/ RRO OMS



Mårten Kongstad
@martenkongstad



Zoran Jovanović
@jovzoran

The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License.
Swedish coat of arms is used according to terms described in the Creative Commons 3.0 Attribution License. (Henrik Dahlström / Riksarkivet Sverige)
Additional graphics by Fredrik Johansson of DCE Studio Nordic.

- Now you know how Android resources work and will hopefully be more comfortable using them in your everyday work knowing how everything runs under the hood.
- You also learned how Runtime Resource Overlay extends the existing pool of resources and how using OverlayManagerService you can access this functionality via a familiar Java interface and it's available in Oreo TODAY.
- Right now this API is not accessible from the SDK and is available to platform builders - OEMs and modders, but all the tools we mentioned, however, are accessible to everyone. Hope you can add this to your toolbox.
- Besides that, what's in it for me? - you may ask.
- For system developers/device manufacturers, it's obvious, you can use this functionality to make your lives easier when customizing and themeing your devices.
- For application developers the choice might not be as obvious though.
- First thing is to understand that you should not try to out-smart the system by e.g. optimizing resource aquisition by e.g. caching your app's resources in a variable or SharedPreferences - handling configuration changes becomes much much harder with literally no gain.
- Next, you should understand that the running system may change some framework resource values in runtime - e.g. a personal theme may be applied - so you have to make sure to use system styles consistently so you don't get funny looking artifacts in your views. Especially, don't try to mix and match a system theme with your own resources like inventing a font color on top of system background - blue text on blue background doesn't really read well.
- If the AOSP patches are an indication of things to come, there will be a time when you will be able to use overlays on your own apps and download them in runtime. We will probably then have a talk on best practices.
- For those inclined to visual arts, there are tools like ThemeCreator that will help you create overlay packages to theme your device.

APPENDIX

References:

- Important files in frameworks/base
 - `libs/androidfw/*`
 - `services/core/java/com/android/server/om/*`
 - `cmd/idmap/*`
- Uploaded, not merged, commits
 - <https://android-review.googlesource.com/#/q/owner:marten.kongstad+status:open>
 - <https://android-review.googlesource.com/#/q/owner:zoran.jovanovic+status:open>
- gradle aapt options:
 - aapt options: [https://google.github.io/android-gradle-dsl/current/com.android.build.gradle.BaseExtension.html#com.android.build.gradle.BaseExtension:aaptOptions\(org.gradle.api.Action\)](https://google.github.io/android-gradle-dsl/current/com.android.build.gradle.BaseExtension.html#com.android.build.gradle.BaseExtension:aaptOptions(org.gradle.api.Action))
 - aapt2 switch for 3.0 preview: `android.enableAapt2=true`
 - aapt2 switch for 3.0, if there are issues and you want to return to aapt: `android.enableAapt2=false`