detectright

NO STONE UNTURNED

Next Generation Device Detection System

Getting started

DetectRight 2.7

Java Edition

Contents

Revision History:

27/Sep/11 – initial draft

10/Feb/12 – v2, release

8/Mar/12 – minor corrections

13/6/13 – major changes for version 2.6

19/7/13 – minor typos fixed

Introduction to DetectRight™

Welcome to DetectRight.

DetectRight is a programming library that allows your program to access a database of tens of thousands of mobile devices and browsers (desktop and mobile), including Connected Devices (or "The Internet of Everything".

DetectRight's core purpose is to convert a Collection of HTTP Headers (or a user agent, device description, device ID or UAProfile URL) to a profile of capabilities of a detected device. This profile is based on your database.

DetectRight improves on the accuracy of other solutions by detecting more signatures, more flexibly. It also stores data at a low level so that fully bespoke profiles can be generated at detection time.

Getting started

1) Install
2) Initialize library with database filename
3) Do detection
4) Use result

DetectRight doesn't need any frameworks and takes normal maps/strings/arrays as input, and so can be used as a component in many different circumstances.

DetectRight uses a compiled SQLite database to get its data, which can optionally (and it is recommended) be copied to a memory instance when appropriate.

Note: This database is optimized for real-time detection purposes, and does not contain the data you would need to populate your own database.

Enterprise and Advanced use cases

An Enterprise use case is using DetectRight to maintain device catalogue lists and keep them synchronised with our data over time. This can also be done by subscribing to our search engine at www.detectright.com/device-search.html or our bulk data services, which also have access to brand and firm metadata, and family group membership.

DetectRight has access to various helper Classes which can be used to perform other useful tasks.

Getting started with DetectRight Java

DetectRight comes as a JAR file, with a class name of com.detectright.core

- JRE 1.6 compatible
- Thread-safe, with critical data shared across the application
- Uses SQLiteJDBC library to access a single data file
- SQLite file can be loaded into memory to increase speed and provide hot updates

Link your project to detectright.jar, making sure your project can also see the SQLite and Apache Commons Codec libraries. You can download these JARs in a zip file from http://www.detectright.com/downloads ("detectright dependencies").

The import is:

```
import com.detectright.core.*;
```

DetectRight, Thread-Safety and Initialization

In a web application or running JVM, DetectRight needs initializing at least once. Since this is a blocking activity, it's best done by running a DetectRight initialization script in a startup object. This allows global options to be set in one place and the database to be loaded into memory at that point.

If you can guarantee DetectRight has already been initialized, then you can skip initialization code on any threads which subsequently access DetectRight. Each thread should have a `DetectRight.resetThread();` command before any detecting is done. This resets thread-specific data.

DetectRight will also work if the option setting and initialization are done on each thread, though only the first thread to execute will end up doing anything meaningful. In this scenario other threads are setting redundant settings and resetting the thread, but not doing any more than that.

A typical initialization command is:

```
try {

        DetectRight.initialize("SQLite///path/to/detectright.data", true);

} catch (ConnectionLostException ce) {

        // DetectRight has lost (or never acquired) a database connection.

} catch (DetectRightException de) {

        // If de.x is an object, it contains an encapsulated exception to examine.
        // It can also be generated for other fatal or semi-fatal errors

}
```

This initializes DetectRight with a link to its database. The second argument says "load it into memory". If this is false then all threads will use a direct link to the file database on disk. This slows down performance and reduces concurrency.

DetectRight Memory usage

DetectRight's memory usage is mostly accounted for by caches, with a core utilization of 20-40Mb for the detection. Note that the SQLite memory data is not stored in heap, but in the process space of the SQLite native library.

Memory usage will also vary according to how many fields are in the database.

Updating DetectRight

For real-time intensive web-app use, it's recommended to load the SQLite database into memory on initialization (see above).

Assuming DetectRight has already been initialized, you would do this to update it:

1) Download new database (see "Downloading a database")
2) Replace old file database (the system shouldn't be locking it)
3) Run this in an unused thread:

```
        DetectRight.resetThread();
        DetectRight.storeDatabaseInMemory("SQLite///path/to/detectright.data");
        DetectRight.close(); // close is now an alias for resetThread() again.

This will also do:
        DetectRight.initialize("SQLite///path/to/detectright.data");
        DetectRight.storeDatabaseInMemory("SQLite///path/to/detectright.data");
        DetectRight.close(); // close is now an alias for resetThread() again.
```

This loads the database into a new memory instance, and when complete swaps them over, locks the application briefly, and then resets the global lookup tables (and soft-resets the useragent cache).

This method is the only way to hot-update DetectRight, since without the memory copy, the application server will lock the database file. In this scenario, the web server would have to be stopped while the file was replaced.

DetectRight and Databases

Downloading a database

A DetectRight database is a personalised snapshot of data from a much bigger database within DetectRight itself.

Some snapshots have very few fields (such as the evaluation database, or databases meant purely for simple analytic detection). Some have many fields. Some have new custom fields.

The database you download is related to your user login at DetectRight.com.

Downloading direct

A new database can be downloaded from:

http://www.detectright.com/download_database.php?druser={username}&drpassword={password}

It can also be downloaded from http://www.detectright.com/cpanel.html when logged in.

If you have a custom database and have enabled data overrides, this download process will slipstream data overrides to your database so that they appear in your data immediately and do not have to wait for a weekly update. This will result in a slightly extended download time.

Express Mode

DetectRight has a flag called “Express Mode”.

DetectRight.setExpressMode(boolean status);

This is set to “true” by default, and affects all threads instantly if changed (unless the thread has isolated its options: for more on this, see “DetectRight Configuration”)

Express Mode On

- DetectRight does less work normalising the user agent though it still looks for and replaces language strings
- DetectRight doesn’t process Accept strings for dynamic information unless there’s no user-agent present
- DetectRight caches a header according to its User-agent and UAProfile URL
- DetectRight doesn’t process the headers for general connection, IP, and other state information (though you can still do these steps independently)
- DetectRight still looks for Opera Mini and other hidden headers for side-loaded browsers
- Other minor shortcuts

Express Mode Off

- DetectRight cleans a URL of proxies, appendages, IMEI and other nasties
- DetectRight processes accept strings to find extra media support to add to the profile
- DetectRight uses the headers to calculate a UID fingerprint and uses that to cache the profile

Express mode is much faster on average in real traffic because the cache hit/miss ratio is better: the caching is at device header level rather than customer header level.

User Agent Cache

The User Agent cache itself is a “ConcurrentRollingCache” object. This is a Map stored in heap which has a configurable maximum item limit, and a buffer of 10% of that size.

Profiles are stored in this cache. When the cache is full, it starts filling the buffer and looking in both for reads. When the buffer is full, it switches the buffer to be the main list, and clears the main list.

Unique user agents generally make up 5% - 10% of any given traffic, so the cache size is important.

Check status of cache:

```
        Boolean status = DetectRight.uaCacheStatus();
```

Enable cache (default):

```
        DetectRight.enableUACache();
```

Disable cache:

```
        DetectRight.disableUACache();
or:     DetectRight.setUACacheSize(0);
```

Reset cache (soft reset: forces the cache to gradually swap over):

```
        DetectRight.resetUACache();
```

Set cache size:

```
        DetectRight.setUACacheSize(100000);
```

Set cache size to unlimited (this will grow heap memory):

```
        DetectRight.unlimitedUACacheSize();
Or:     DetectRight.setUACacheSize(-1);
```

Get current cache size:

```
        int cacheSize= DetectRight.getUACacheSize();
```

DetectRight speed, memory and concurrency.

One of the challenges of optimising DetectRight is in overcoming the drag factor of the SQLite dependency: there's a direct link between the average response time at any given second and the weight on the database engine. It's on the roadmap to remove this dependency, but for the moment it's the only way for DetectRight to get its super-accurate results.

DetectRight uses a shared JDBC link across all threads: which has to be synchronized (one thread at a time can do SQL queries). This creates a bottleneck.

The way we mitigate this is:

1) Encourage the loading of the database into memory (this avoids extra file access and locking which would slow an individual query down even more).
2) Use a top-level user-agent cache (this avoids triggering the detection process entirely for 85-95% of detections) – default is 100,000 detections, and the cache involved is gentle when it rolls over.
3) Use an SQL query cache in heap – avoid trips to the database for the same query: this improves concurrency a when the system is under load by avoiding blocking on an SQL query. We use a 10,000 query cache. There was very little practical benefit to increasing this to, say, 50,000: it used more memory but had a minimal effect on performance. In fact, there was a very slight drag because it's marginally quicker to look up a result in a 10,000 row map than a 50,000 row map.

How does DetectRight perform in an app server environment?

Our tests were done on a standard VM on the Cloud provider GoGrid which was running Windows Server 2008R2, and Tomcat 7.

We created a load tester which re-processed traffic logs and threw them in a controlled multi-threaded way at the Tomcat installation, while monitoring with YourKit (http://www.yourkit.com – recommended!).

The load was generated with a maximum of 50 threads generating requests with an random interval of between 10 and 20ms. On the server this was experienced as a load of between 60 and 239 accesses per second.

Our tests showed that with the default configuration above, DetectRight was using up to 150Mb consistently, and that this remained stable over time. Desktop-centric datasets generated less load and memory than mobile-centric ones.

The average CPU load for this load was < 5%. As a baseline, the average CPU for a JSP script which merely copied the request headers into a map and then outputted "OK" was 3%.

The average servlet response time was < 1ms (the average was too small for YourKit to measure and tended to zero over time).

This response time was not impacted by using "storeDatabaseInMemory" to update the database in-situ.

Doing detections

DetectRight offers a number of functions to get profile data, which is always returned as a Map<String,Object> (internally it's a LinkedHashMap).

Detection with Headers (Full or partial Map of key/value pairs)

If you're using JSP, then you have to copy the request headers to an appropriate map before passing that map off to DetectRight.

```
<%@ import="com.detectright.core.*, java.util.*" %>

<%

 Enumeration enames;

   Map<String,Object> lhm = new LinkedHashMap();

   enames = request.getHeaderNames();

   while (enames.hasMoreElements()) {

      String name = (String) enames.nextElement();

      String value = request.getHeader(name);

      lhm.put(name, value);

   }

Map<String,Object> result = DetectRight.getProfileFromHeaders(lhm);
```

This is the most thorough way to detect, because DetectRight can use other headers such as the Accept headers or language preferences to do other work, and can even find a unique user ID if there's one lurking (assuming Express Mode is off).

This function uses adaptive detection, in that component version changes such as browser and OS can change the capabilities (not just the metadata such as "browser version").

Detection with a User Agent

```
String useragent = "Nokia3510i/1.0";
Map<String,Object> result = DetectRight.getProfileFromUA(useragent);
```

Although not as thorough as getProfileFromHeaders, this still detects adaptively: for instance, if the browser in the user agent is different to the one shipped with the device, the profile will swap out the old browser with the new one. Other component changes such as OS may also be detected.

Note that the header detection is capable of looking in the headers to find "hidden" user agents, but this ability is lost if you're feeding DetectRight only a user agent.

Detection with a UAProfile URL

This is a simple lookup between the URL and the model number it's mapped to in the DetectRight system. The profile acquired will be the "base" profile of the device: i.e. "factory spec".

```
String uap = "http://nds1.nds.nokia.com/uaprof/N6303iclassicr100.xml";
Map<String,Object> result = DetectRight.getProfileFromUAProfile(uap);
```

Detection from a TAC code

TAC stands for "Type Allocation Code", and is the first few digits of the IMEI number of a device. It's the nearest thing the industry has to a unique ID for a model, as it's given to the device as it passes through approval bodies such as the FCC. One "model" may have many TACs. DetectRight contains many TACs, though the list is not exhaustive, and is not currently maintained, much as we'd like it to be.

```
Map<String,Object> result = DetectRight.getProfileFromTAC("01010101");
```

Detection by device name/type

DetectRight can also retrieve a device from a manufacturer/model lookup: which produces a basic "factory" profile.

```
Map<String,Object> result = DetectRight.getProfileFromDevice("Device", "Nokia", "3510i");
```

DetectRight uses a concept of "Entity type". DetectRight's data is not limited to mobile devices: it can handle any kind of object. If no entity type is passed, DetectRight checks a subset of entity types which are "things" (such as Tablets) as opposed to components (such as Browsers).

For instance, the data shipped with DetectRight contains data for Browser, JVM, OS, Developer Platform, Tablet, Device (the default entity type for mobile devices), PDAs, and others. This entity type is key to making identity fields such as "is_mobile_device" and "is_tablet" work, because they're keyed to Entity type in the database. That means that they are always populated, and follow any changes to the entity type.

Detection from a device ID

The combination of entity type, manufacturer and model can be calculated into a 32 character MD5 checksum called a "Device ID". The algorithm to do this is kept in the EntityCore.makeHash method.

This example:

```
String entityhash = EntityCore.makeHash("Device","Nokia","3510i");
Map<String,Object> result = DetectRight.getProfileFromDeviceID(entityhash);
```

Is functionally identical to this:

```
Map<String,Object> result = DetectRight.getProfileFromDevice("Device", "Nokia", "3510i");
```

DetectRight has a large list of aliases in its database through which it does these lookups.

Fields in the detection

This will depend upon the database you've downloaded.

The default database comes with a very small selection of data points, but this can be greatly expanded by upgrading your account.

A simple list of the fields in your database can be acquired like this:

```
List<String> fieldNames = DetectRight.getCapabilities();
```

And for more detail:

```
Map<String,SchemaPropertyCore> fields = DetectRight.getAllFieldNames();
```

Key = lower case field names.
Value = associated SchemaPropertyCore objects

The SchemaPropertyCore class is the main object for storing the details behind each field in the database.

Not included in this field list are some bonus metadata fields returned with every profile. These are diagnostic or information fields such as "components" which contains the list of components detected, or "internalid" (database ID of detected device, if any) and are inserted at the final stage of processing.

Getting a list of fields

Another way of getting a list of fields but in Map form is this:

```
        Map<String,Object> map = DetectRight.getAllFieldNames();
```

It returns a map keyed by (lower case) field name, with each one having a further map lifted from the "schema_properties" table in the SQLite database.

"display_name" is the field which is displayed in the output if DetectRight.strictExportNames is set to true, and preserves the capitalisation.

"output_map" is the path to a node in our QuantumDataTree which retrieves the actual data, and "validation_type" either runs the value through one of the built-in validators, or through a content validator which runs off the schema_property_values_table, whose job is to clean, validate and generate extra data.

For more information about the SchemaPropertyCore object, see the Javadoc at http://www.detectright.com/javadoc/SchemaPropertyCore.html

Assorted questions

Why is DetectRight static?

DetectRight was created for static invocation to avoid having to pass DetectRight instances around the program into virtually every method: since DetectRight is an API more than an object.

DetectRight has three types of preferences (there is a tablet later on in the document):

1) Thread-only (related directly to the detection or state of the thread)
2) Thread/Application (global variables which might need to be overridden on an individual thread).
3) Application (global variables holding critical application information).

Is DetectRight Thread-Safe?

Yes. There are certain functions in the program that are synchronized: mostly to ensure initialization doesn't happen too many times. The SQL query command in DetectRight's database engine is synchronized, which means that loading the database into memory and caching the occasional object is mandatory for speed.

How fast is DetectRight?

DetectRight is pretty fast, especially going from user-agent cache.

The biggest impact on speed comes from where the database is (memory, local, network drive). If the SQLite file is being accessed directly across a network, you can expect terrible performance, and local drive is better, but loading into memory is always the best option.

The actual speed also varies according to the size of the database you're feeding it. Part of DetectRight's custom database service is allowing a company to have a cut-down schema (to increase speed or decrease memory usage), or even a schema with items from other device repositories (for instance, mixing WURFL and DeviceAtlas style fields in the same schema). This allows the database to be exactly suited to the use case.

However, DetectRight is simply not as fast as WURFL or DeviceAtlas overall, because it is doing a lot more work. DetectRight uses lots of cache strategies to improve performance in real traffic, and can handle a large amount of traffic, but it does not compromise on accuracy.

Can I enter my own data?

Customers with the premium data upgrade or an Enterprise licence can, through a web interface control panel. This allows a client to attach alternative values to devices that are guaranteed to appear in the profile whenever a user agent is detected.

How accurate is the data?

It's best effort. Although we cannot guarantee the accuracy of the data, the processes it's gone through means that it's more accurate than anything else (probably).

Testing DetectRight

System Status Panel

There are numerous functions built into DetectRight that are immediately usable for testing it. For instance, the "DetectRight.status()" method returns an List of diagnostic information.

```
System.out.println(Functions.implode("\n",DetectRight.status()));
```

Detection Diagnostic Test

Also, there's a detection function built in which will test a useragent, and generate a rich diagnostic of the entire detection and how it was managed. This is useful diagnostic information to send back to us if there's a detection you're particularly worried about.

```
List<String> lst = DetectRight.testDR("Alcatel-ELLE-N3/1.0 Profile/MIDP-2.0
Configuration/CLDC-1.1 ObigoInternetBrowser/Q03C");
```

Detecting user agents in bulk

To save you writing the code, DetectRight can also take in a list of useragents or 32-character device IDs and output a detection with all of the fields in the database, along with some diagnostic information.

```
DetectRight.testFile("c:\\data\\test\\ua.txt", "c:\\ data\\output.txt", "UA");
```

You can also pass a Scanner object as input, and a Writer object as output, just in case you need to pipe your detections somewhere esoteric. The output that comes out is a tab delimited text file, with field headers. The process will also generate additional diagnostics including a mean detection time to System.out.

Note that processing a unique list of user-agents will result in performance timings that are not representative of performance in real-traffic. If you would like a free sample of "real" traffic to run against a test, please enquire. Also, note that the average timings will depend on the size of the user agent cache, and the mobile/non-mobile makeup of the traffic.

DetectRight and Catalogue Management

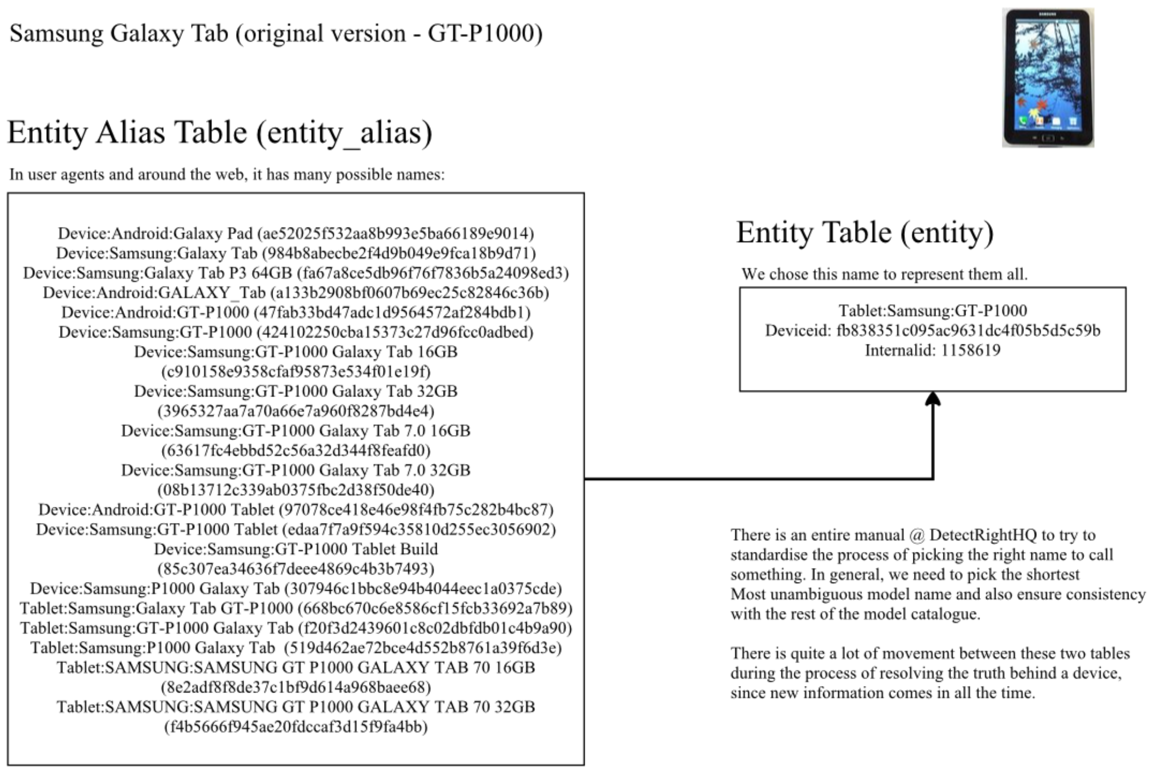
DetectRight has some API functions enabled for the Enterprise version which allow synchronization with an external database, and extraction of data.

Important: extraction of the data directly from the SQLite file is not permitted by the standard DetectRight Data and Web Services Licence governing the device database download.

Understanding DetectRight's catalogue

Figure 1 shows an example of the Samsung GT-P1000: the original Galaxy Tab 7.0, and how it's represented in DetectRight.

What it shows is that there are multiple possible names for a device, and DetectRight tries to store as many of them as possible. One of those names is the "official" name (official to DetectRight, that is). All of the possible names resolve to the "official" name, which is what appears in the manufacturer and model fields in a detection.



Samsung Galaxy Tab (original version - GT-P1000)

Entity Alias Table (entity_alias)

In user agents and around the web, it has many possible names:

Device:Android:Galaxy Pad (ae52025f532aa8b993e5ba66189e9014)
Device:Samsung:Galaxy Tab (984b8abecbe2f4d9b049e9fca18b9d71)
Device:Samsung:Galaxy Tab P3 64GB (fa67a8ce5db96f76f7836b5a24098ed3)
Device:Android:GALAXY_Tab (a133b2908bf0607b69ec25c82846c36b)
Device:Android:GT-P1000 (47fab33bd47adc1d9564572af284bdb1)
Device:Samsung:GT-P1000 (424102250cba15373c27d96fcc0adbed)
Device:Samsung:GT-P1000 Galaxy Tab 16GB (c910158e9358cfaf95873e534f01e19f)
Device:Samsung:GT-P1000 Galaxy Tab 32GB (3965327aa7a70a66e7a960f8287bd4e4)
Device:Samsung:GT-P1000 Galaxy Tab 7.0 16GB (63617fc4ebbd52c56a32d344f8feafd0)
Device:Samsung:GT-P1000 Galaxy Tab 7.0 32GB (08b13712c339ab0375fbc2d38f50de40)
Device:Android:GT-P1000 Tablet (97078ce418e46e98f4fb75c282b4bc87)
Device:Samsung:GT-P1000 Tablet (edaa7f7a9f594c35810d255ec3056902)
Device:Samsung:GT-P1000 Tablet Build (85c307ea34636f7deee4869c4b3b7493)
Device:Samsung:P1000 Galaxy Tab (307946c1bbc8e94b4044eec1a0375cde)
Tablet:Samsung:Galaxy Tab GT-P1000 (668bc670c6e8586cf15fcb33692a7b89)
Tablet:Samsung:GT-P1000 Galaxy Tab (f20f3d2439601c8c02dbfdb01c4b9a90)
Tablet:Samsung:P1000 Galaxy Tab (519d462ae72bce4d552b8761a39f6d3e)
Tablet:SAMSUNG:SAMSUNG GT P1000 GALAXY TAB 70 16GB (8e2adf8f8de37c1bf9d614a968baee68)
Tablet:SAMSUNG:SAMSUNG GT P1000 GALAXY TAB 70 32GB (f4b5666f945ae20fdccaf3d15f9fa4bb)

Entity Table (entity)

We chose this name to represent them all.

Tablet:Samsung:GT-P1000
Deviceid: fb838351c095ac9631dc4f05b5d5c59b
Internalid: 1158619

There is an entire manual @ DetectRightHQ to try to standardise the process of picking the right name to call something. In general, we need to pick the shortest Most unambiguous model name and also ensure consistency with the rest of the model catalogue.

There is quite a lot of movement between these two tables during the process of resolving the truth behind a device, since new information comes in all the time.

Figure 1 - the many faces of the GT-P1000

More about Device IDs and descriptors

Each of the names for a device in DetectRight generates a Device ID.

In DetectRight, Device IDs do not refer to useragents like in WURFL. A device ID is directly calculated from an actual device name: a combination of an Entity type, a Category (which would be the manufacturer for devices), and a Description (the model name, usually).

DetectRight calls these names "Descriptors". They're colon-separated strings of the following format.

Entity type:Category:Description:Subclass:Major Revision:Minor Revision:Connection:Build

e.g. Device:Nokia:3510i:AT&T:1.0:1.01:3G:AGH/F2

The Device ID is worked out by this code:

```
        String descriptor = new String(entitytype + "/" + category + "/" + description);

        String deviceID = Functions.md5(descriptor.toLowerCase());

Or, to save you doing it:

        String deviceID = EntityCore.makeHash(entitytype,category,description);
```

A "thing" in DetectRight has many possible names, and thus Device IDs.

At DetectRight HQ, we choose one of these to be the primary device ID (which tends to be chosen for consistency with a manufacturer's model naming strategy: for instance, Alcatel OT- is changed to Alcatel One Touch, and Samsung models all have their prefix of GT-, SGH-, SPH-, SCH-, SHW- or YP- (with occasional exceptions).

Catalogue Management Functions

getAllDevices – Data dump

Invoked without arguments, this returns everything in the "entity" table in the database in a Map or Maps, keyed by DeviceID. This is a large, time consuming process and should not be done repeatedly. Despite the name, this dump is not limited to Mobile Devices, but Tablets, Browsers, and anything else in the database.

```
        Map<String,Object> devices = DetectRight.getAllDevices();
```

If you pass a List<String> of Device IDs, it will confine its results to those. This facility is used in conjunction with the `DetectRight.getDeltaDeviceIDs` method (whose job is to extract from the databases devices you don't have yet). An null will result in everything being returned (and is equivalent to calling it without arguments), an empty list will return in an empty result set.

```
        List<String> deviceIDsIHave = new ArrayList<String>();

        // next command will return an empty map if the list remains empty

        Map<String,Object> devices = DetectRight.getAllDevices(deviceIDsIHave);
```

getDeltaDeviceIDs – Get a list of what you haven't got (yet)

This method was put in to help synchronize DetectRight with existing catalogues that might be shadowing DetectRight's catalogue. If you feed in a list of device IDs you have, you get back a list of the ones you don't have.

At least, you get back a list of the primary/official IDs you don't have.

```
List<String> oldDevices = new ArrayList<String>(); // this should actually contain device IDs
you have in your own catalogue.

List<String> exceptions = new ArrayList<String>(); // we're passing an empty

Map<String,String> remaps = new HashMap<String,String>(); // another empty list.

List<String> newDeviceIDs = DetectRight.getDeltaDeviceIDs(oldDevices,exceptions, remaps);
```

After this call, newDeviceIDs will contain a list of device IDs that aren't covered by your incoming list. This can be passed to getAllDevices.

In addition to this, the Exceptions list will return a list of device IDs which no longer exist. This means they've been determined by us to be invalid and have been removed from the working set.

The map "remaps" contains a list of your old device IDs, and what they map to in the new database. This covers the situation where devices have been merged or separated, or where a previous primary name has become an alias.

We're sorry we can't make the process less messy: but then, the whole process itself is messy (making sense of it is probably the major part of our workflow).

getDeviceIDMap: Keeping up to date with aliases

If you've got a device ID and you want to get the whole picture on what it is now:

```
        String oldDeviceID = "ae52025f532aa8b993e5ba66189e9014";

        List<String> aliases = new ArrayList<String>();

        String currentID = DetectRight.getDeviceIDMap(oldDeviceID, aliases);
```

After this, `currentID` will contain the current primary Device ID of the Samsung GT-P1000, and `aliases` will contain the list of alternate/secondary device IDs.

getCatalog: Devices per manufacturer

```
Map<String,String> nokiaDevices = DetectRight.getCatalog("Nokia");
```

Key = device ID

Value = model name of device

DetectRight and Change Management

The “hash” field of the main entity table is the Device ID representing the primary name.

The entity_alias table contains all of the other possible IDs, and these refer back to the same “thing” (Device/Tablet/Browser/whatever).

Unfortunately, reality is complicated. If a device enters the database with one name (for instance, a codename, an FCC designation, or a marketing name), it might end up later with a different cleaner one as the primary name, or it might have to be merged with another device. This would mean that your database had a different device list from update to update.

A Device ID is almost never lost

DetectRight is built so that once a device ID (device name) has been generated, it is never lost, except when it has been determined that it was rubbish in the first place. When a device is renamed or merged, the old Device ID is recorded (in the table entity_alias) as a pointer.

DetectRight will always return a valid result (for instance, from `getProfileFromDeviceID` for a device ID that you might have got from a previous database. However, it might now be attached to something more specific, or another device (but ONLY if that match is better than the previous one: you will NEVER get a situation where a device ID is attached to an inappropriate device).

Since the Device ID is a function of the actual name, the idea is that data only gets more accurate.

I’d note that devices at DetectRight HQ go through a number of statuses at HQ before hitting the live data, so as to try and minimise the number of changes once a device goes “live”.

What devices don’t I have yet?

The question “what’s in the new update that I don’t have yet?” is non-trivial, and requires comparing the Device IDs you have with both the primary list and the aliases/secondary IDs.

Here’s the code which would give you a list of devices that you didn’t pass in:

```
List<String> deviceIDsIHave = new ArrayList<String>();

List<String> newDevices = DetectRight.getDeltaDeviceIDs(deviceIDsIHave);

// next command will return everything.

Map<String,Object> devices = DetectRight.getAllDevices(newDevices);
```

For keeping up to date, it’s handy to know when things have changed their primary Device ID in DetectRight, so you can follow the changes.

This version of the getDeltaDeviceIDs function allows you to map catalogue changes:

```
List<String> exceptions = new ArrayList<String>();

Map<String,String> remaps = new HashMap<String,String>();
```

After this next command, `exceptions` will contain device IDs that no longer exist (this should always be empty!), and remaps will contain a list of device IDs passed in that are now mapped to something else.

```
	List<String> newDevices = DetectRight.getDeltaDeviceIDs(deviceIDsIHave, exceptions, 
remaps);

	Map<String,Object> newDeviceData = DetectRight.getAllDevices(newDevices);
```

You now have a Map of new device data, and a list of the IDs in your original list which have been remapped.

This combined information can be passed to one of your internal processes to make appropriate changes to your own device database.

DetectRight Configuration

DetectRight has many configuration options which are explained in the table.

Adaptation vs Exception (see “deviceNotFoundBehavior”)

What does DetectRight do when the detection doesn’t connect with a main database record?

An example of where is meaningful is for a desktop browser detection. In this case, although DetectRight can identify the browser and version (and construct a profile with appropriate values), there is no underlying identifiable device.

Equally, a detection from a new HTC android phone might generate a profile based on the version of Android, and even work out the model’s name (for instance, “Android SuperDuper”, but that might not correspond to a device shipped with the database.

In both of these detection cases, if the detection doesn’t not correspond to a “nominative entity” (i.e. something which gives the detection its name: like “a Nokia 3510i” or “A Samsung GT-P1000 tablet”), then an exception will be generated if deviceNotFoundBehavior has been switched to “Exception”. In the case of an adaptation, the system will provide a profile based on what it’s found and return that. This is the default.

Summary:

Adaptive results (no exceptions): default, turned on by:

```
DetectRight.adaptiveProfileOnDeviceNotFound();
```

Exceptions:

```
DetectRight.generateExceptionOnDeviceNotFound();
```

Other Cool stuff

Fun with Headers

There's a lot going on in DetectRight, and the HTTPHeadersCore object includes a lot of convenience functions for working with headers: for instance, getting a profile from an accept string, cleaning URLs, removing language strings, generating UIDs (user IDs), etc.

Fun with PHP

There's a partial PHP compatibility library ("Functions") built into DetectRight which recreates some useful PHP functions in Java and C#. It also supports serialization and deserialization with limitations. Serialization doesn't support objects (though deserialize does), mostly due to the faff surrounding null bytes. While it tries its best to support references and reference counting, the whole process is so poorly documented in PHP that we work around it by not using PHP structures with recursive references.

Fun (?) with Quantum Data Trees

Quantum Data Trees are an object built into DetectRight that allows nested trees to be created for arbitrary data at varying degrees of confidence (and also allows quantitative data points to do so), and then be queried back across the tree structure. It's pretty powerful for representing all sorts of data.

Fun with Profiles

Detectright has an object called "DRProfile", which is still in beta. It essentially encapsulates a detection, turning the detection process of DetectRight.getProfileByXXX into one which treats as detection as an object: this allows cool stuff such as being able to insert overrides into an detection, get the confidence level/importance of any given data point, check for various component types, and much more. Plus it's handy having a detection encapsulated as an object rather than a map, since the metadata is much better structured. We will be making more use of this ourselves later, but for speed reasons, not as a main use case.

DetectRight Configuration flags

DetectRight configurations can affect different levels of DR behaviour. The normal behaviour is for all threads to inherit the global properties. In this configuration, “options” is set to “Global”, and is a thread-level variable.

To make your thread use its own option set, do:

```
        DetectRight.useThreadBasedOptions(true);
```

Any settings set after this command (assuming the selected option can be used as a Thread option) will affect only the running thread.

Certain options are set at application start-up and will have no effect if changed later (such as the connected database). This is because they are read during initialization, and this only happens once.

If a property is requested, the following logic is used:

1) Is this thread using thread-specific configuration?
   ```
           if (DetectRight.usingThreadBasedOptions()) {}
   ```
2) Have we set a valid value for this property for this thread? (if yes – return property)
3) Do we have a global value for this property (if yes – return property)
4) If no, return hardcoded default.

DetectRight Configuration Flag Reference

Property Name	Description	Related Functions	Visibility	Default	Valid Values

`options`	Dictates whether the thread looks for thread-specific settings where appropriate.	`void useThreadBasedOptions(boolean status)`
		`Boolean usingThreadBasedOptions()`
			Thread	false	true
					false

`expressMode`	Dictates whether DetectRight takes shortcuts in detection	`void setExpressMode(boolean status)`
		`Boolean getExpressMode()`
			Thread
			Global
				true	true
					false

`enableUserAgentEntityType`	Determines what appears in fields relating to manufacturer and model in the event of a "failed" detection.
	If true, manufacturer is set to "UserAgent" and the model is set to the same as the user agent passed in.
	If false, manufacturer is set to "Generic", and the model reflects the type of device (e.g. "Desktop").
		`void setEnableUserAgentEntityType(boolean status)`
		`Boolean getEnableUserAgentEntityType()`
			Thread
			Global
				false	true
					false

`readersAsTablets`	If something is an e-Reader, does DetectRight report it as a tablet?	`void setReadersAsTablets(boolean status)`
		`Boolean getReadersAsTablets()`
			Thread
			Global
				false	true
					false

`throwOutGenericDevices`	If manufacturer equals "Generic", do we throw a DeviceNotFound Exception?	`void setThrowOutGenericDevices(boolean status)`
		`Boolean getThrowOutGenericDevices()`
			Thread
			Global
				false	true
					false

`DIAG`	Diagnostic mode flag. If true, DetectRight generates diagnostic timestamped data into a map of checkpoints, and outputs voluminous	`void setDiag(boolean status)`
		`Boolean getDiag()`
			Thread	false	true
					false

	diagnostic information to Stdout. For debugging only.				

`startTime`	For diagnostics. The base time from thread initialization to fill in diagnostic checkpoints.	`void setStartTime(double start)`
		`double getStartTime()`
			Thread	Set when DetectRight is initialized	

`prevTime`	The timestamp of the previous checkpoint.	`void setPrevTime(double prevTime)`
		`double getPrevTime()`
			Thread	Set automatically at each checkpoint	

`maxCheckpointLevel`	Different checkpoint statements in DetectRight's code have different levels. This sets the maximum that will be displayed, allowing some granularity in the output.	`void setMaxCheckpointLevel(int level)`	Thread	999	Any valid integer

`redetect`	Setting this to true will defeat all cache reads in the thread	`void setRedetect(boolean status)`
		`Boolean getRedetect()`
			Thread	false	true
					false

`overrideHighlight`	When detecting, if this is true, a single character prefix will be added onto the value string to designate whether it's a user-specified override, and how it relates to the previous value.	`void setOverrideHighlight(boolean status)`
		`Boolean getOverrideHighlight()`
			Thread	false	true
					false

`flush`	If set to true, all caches encountered in the system are reset. It is not recommended to do this.	`static void setFlush(Boolean status)`
		`static Boolean getFlush()`
			Thread	false	true
					false

`userAgentsAsBrowsers`	If set to true, promotes the detected browser to fill manufacturer and model fields.	`static void setUserAgentsAsBrowsers(Boolean status)`
		`static Boolean getUserAgentsAsBrowsers()`
			Thread
			Global
				false	true
					false

deviceNotFoundBehavior	If set to "adaptation", DetectRight returns an adapted profile even if the root device detected isn't in the database. If set to "exception", DetectRight will generate a "DeviceNotFound" exception if the detected device doesn't exist in its tables.	static String deviceNotFoundBehavior() static void generateExceptionOnDeviceNotFound() static void adaptiveProfileOnDeviceNotFound()	Thread Global	Adaptation	Adaptation Exception
accessLevel	Internal integer to specify the access level of the logged on user. 8 or above specifies superuser. Not useful in normal DR operation.	static void setAccessLevel(int level) static int getAccessLevel()	Global Thread	1	Any positive integer. Increasing numbers indicates increasing access level.
data_owner	Used to store and differentiate different levels of data ownership of entities, profiles and overrides. Should not be used or needed.	static void setDataOwner(String dataOwner) static String getDataOwner()	Global Thread	System	Arbitrary string
strictExportNames	For mixed-case fieldnames (such as "displayWidth"), setting this to true preserves the original case of the field. Setting to false makes it lower-case.	static void setStrictExportNames(Boolean status) static Boolean getStrictExportNames()	Global Thread	true	true false
forcePortraitTablets	Tablets in DetectRight are stored in our main database with landscape resolutions, though usable screensize is usually portrait. Setting this	static void setForcePortraitTablets(Boolean status) static Boolean getForcePortraitTablets()	Global Thread	true	true false

	to true forces all tablet dimensions (including usable screensizes) into portrait.				

`flat`	Internal flag to check the database for “flat trees”. This is a way of storing data in DetectRight tree structures that’s more “pre-canned” than the usual tree structure. This should generally be set to true and left there.	`static void setFlat(Boolean status)`
		`static Boolean getFlat()`
			Global
			Thread
				true	true
					false

`LOG`	Internal flag to specify whether DetectRight should be logging or not.	`static void setLog(Boolean status)`
		`static Boolean getLog()`
			Thread	false	true
					false

`LOG_METHOD`	Enum to determine where log output should be sent.	`static void setLogMethod(LOG_METHOD lm)`
		`static LOG_METHOD getLogMethod()`
			Thread	ECHO	LOG
					ECHO
					BUFFER

`logQueries`	Determines whether SQL run should be written to the checkpoint collection. Will have no effect if DIAG is off.	`static void setLogQueries(Boolean status)`
		`static Boolean getLogQueries()`
			Thread	false	true
					false

`username`	What username this is running as. Unused.	`static void setUserName(String username)`
		`static String getUserName()`
			Thread	Anonymous	Any string

Appendix A – Data fields in the default DetectRight database

brand_name	Manufacturer, or (is there isn't one identifiable) other data depending on configuration flags
device_dp	Developer Platform of the device (e.g. Android, Series 40, Windows Mobile), expressed as manufacturer/description
device_dp_version	Version of developer platform
device_os	OS installed (note that for Windows Mobile devices, this is Windows CE). Other values include "iphone OS", "BlackBerryOS" and "Android".
device_os_version	Version of OS
device_type	The type of device: for instance, "Device" (mobile phone), "PDA", "PMP", "Tablet", "STB", "Games Console", "Handheld Console"
diagonal	Diagonal screen size in inches
is_tablet	Is this an Internet Tablet? Note that DetectRight has a cut-off of a 6.5" diagonal screen size for the tablet classification: below this, something is a "PMP" (Personal Media Player) if it can't hold a SIM, and "Device" if it can.
is_wireless_device	Can this detection be classified as being from a "wireless" device?
mobile_browser	Detected browser (manufacturer and name of browser: note that with Openwave this is "Openwave Openwave", which is technically correct!).
mobile_browser_version	Version of detected browser.
model_name	Name of model, if known. If no model name is detected, contents will depend on the value of the configuration flags "enableUserAgentEntityType" and "browsersAsDevices" (see DetectRight configuration flag reference)
resolution_width	Physical main screen width in pixels
resolution_height	Physical main screen height in pixels
components	Comma separated list of components that made up this detection, expressed in DetectRight descriptor format (e.g. "Chipset:Qualcomm:MSM6200"). Browser, OS and DP version are acquired from this list.
altids	32 character checksums of alt_descriptors
devicedescriptor	DetectRight descriptor for this device based on internal device type, manufacturer and model (e.g. Device:Nokia:3510i). May also contain version information.
alt_descriptors	DetectRight descriptors for the various aliases for the device DetectRight contains.

deviceid	32 character checksum based on device descriptor.
internalid	The database ID of the detection: if this is zero (for instance, in a desktop detection), then DetectRight has produced a custom profile based on the browser, OS, etc. It’s possible for DetectRight to detect a manufacturer and model from the useragent but have no record of it in the shipped database. In that case, a value of zero in this field tells you that DetectRight is using defaults to populate the fields.

Background: How DetectRight deals with data and fields

In most databases, the set of data which is acquired for the database is in the same format and covering the same information as the output. This makes sense: why capture data you haven't got a space for?

And yet, there is so much possible information about devices that it's often the case that you need to populate a new field quickly. So it makes sense to acquire information from multiple sources, and store it until it's needed later.

Most device databases also like to think that whatever data they don't contain is unimportant or irrelevant. However, if you look at the schemas of BrowserCaps, UAProfile, WURFL, DeviceAtlas, W3C Basic, and DetectRight's own schema, you can see just how little overlap there actually is: especially at the edges. Even when two data points cover similar ground, they might be expressed differently: for instance, UAProfile signals MP3 support by putting "audio/mp3" in the ccppaccept field, yet DeviceAtlas has an "mp3" boolean field, and DetectRight has a "hasmp3" field. If you have a piece of information such as "the maximum permitted bitrate of an MP3 on this device" (a perfectly useful piece of information), the existing containers for data simply don't have a space for that, and would express it differently if they had.

The situation is even messier in the world of human readable specification data such as that released by manufacturers: lists of stuff which to be fully exploited need to be translated into machine-readable data points.

DetectRight developed QuantumDataTrees (and their companion, QuantumDataCollections) to be able to store any information about anything, anywhere, and to be able to reconcile conflicting items of information numerically.

Features of a Quantum Data Tree

1) Uses a path/node and data point syntax structure that allows any data to be represented, and allows hierarchies of properties
2) Reconciles different on/off information or conflicting numeric data points based on a combination of "status" (1/0), "importance" (an integer), and "direct hit" (whether a piece of data is implied or asserted).
3) Rich querying allows searching across the tree at different levels with wildcards to find instances of objects, and to return lists, scalars or Booleans.
4) Can contain metadata and objects at each node addressable by querying
5) Multi-branch trees can be combined seamlessly: meaning that you can overlay the properties for a browser onto the properties for a device and watch the tree make sense of the resulting profile.
6) Tree can be serialized to a path list array for storage

Examples

A piece of data in a Quantum data Tree is represented by a path and a payload.

For instance, if we want to express MP3 support, we can write this:

Media//Player//File Format:Audio:MP3//status=1

But what if we wanted to say whether a device supported ID3 v2? We would write:

Media//Player//File Format:Audio:MP3//Audio:Metadata:ID3::2.0//status=1

This is one item of information: but represented like this, we can see that we have a lot of implied data:

hasaudio = true

hasmedia = true

hasplayer = true

hasmp3 = true

hasaudiometadata = true

hasid3 = true

hasid3v2 = true

If we want to put a non-tree related piece of data into the tree:

Display//dimension=size{value:128x96;units:pixels;importance:50}

Defines a screensize for the display. But what about the secondary display?

Display//1//dimension=size{value:128x96;units:pixels}

What about the camera?

Media//Recorder//Camera//dimension=size{value:128x96;units:pixels;importance:50}

Media//Recorder//Camera//dimension=size{value:3x2;units:inches;importance:50}

This also implies that the device actually has a camera, so any query asking that would be triggered to true by the presence of this data point.

Why Quantum?

It's not just a buzzword. A Quantum Data Tree stores a collection of possible states against each node, and a collection of possible values against each data point in a Quantum Data Collection.

These states are then superimposed on each other at query time and collapsed down to a single observable value: so the tree only collapses its probability wave when it's observed.

This allows us to collect data freely from different heterogeneous sources and have it reconciled in real-time. It also means that data which is implied by, or dependent on another data source lower in the tree will always be present: this removes a lot of the potential problems in a workflow that could lead to this:

Has audio=0

Has mp3 = 1

This might be perfectly possible in other schemas where the manual workflow demanded that both fields be maintained independently.

Hierarchical metadata

Each node has a separate metadata Map: this can be configured so new data is only added to the child node if it's not present in its parents: allowing the kind of data cascade seen in a WURFL tree: and indeed, a Quantum Data Tree is capable of containing a WURFL tree purely by virtue of its parent/child relationships and its metadata.

OK, but why does this matter?

Because with appropriate mapping, it allows us to harmonise all device data, everywhere, by making sure it all says the same thing, in the same way. This allows us to then export to any schema from the tree and have it consistently delivered (courtesy of various validators in and out to handle formatting issues). This means that the same master tree can deliver data for WURFL data points, W3C data points, UAProfile data points, and also take data entry in any or all of these formats from any source.

Decoupling the data from the schema, and the data engine from the detection engine ensures that you can improve your detection and data without moving to another set of data. Just because a particular set of data points is useful doesn't mean that the detection engine or API based on that is best-of-breed.