METASPEX

Metaspex Technical Insights

VIII - Configuration

Introduction

As Metaspex features the direct expression of business ideas, with a maximum of expressiveness, conciseness and performance, all at the same time, users will reasonably expect fine control over the implementation of their business logic.

This is why Metaspex applications typically leverage user-written directives in one or several configuration files. These directives can drastically change the behavior of the application. As an example, the same high-level requirements, expressed with the Metaspex framework, can operate on any one of the supported databases.

This means that the same compact binary, capable of containing hundreds of web services and their associated behaviors, alongside large ontologies, all fitting into a few tens of megabytes, can run on different databases, provided it is deployed with different configuration files during implementation.

Similarly, the same binary can implement vastly different security policy behaviors, from open organizations allowing any user to join, to closed ones where two-factor authentication is activated.

The purpose of this technical insight is to offer a glimpse of the wide variety of directives Metaspex supports in a configuration file. We cannot show them all—there are approximately 200 of them. We'll cover some of the most important ones. The Metaspex reference guide details the rest

Naming and location

Metaspex expects the configuration file to be called **hx2a.conf**. It will find it at the location pointed to by the **HX2A_CONF_PATH** environment variable. A typical value for that variable is "." (the current directory), meaning that the application will find the configuration file where it is started.

Comments

Comments in configuration files start with the character “#”. All characters that follow on the same line are ignored. Here are a few examples:

```
#
# My configuration file.
#
domain "mydomain.com" # This bit won't matter.
```

Keywords and JSON arguments

A configuration file contains very simple structures, first a keyword, then zero, one or several JSON values separated by whitespace characters.

As an example, this is a line in a built-in configuration file delivered with the Metaspex distribution:

```
index     "directory" "hx2araw_org__code" "@o" "h:k"
```

We’ll cover what it does later in this document, but for now, suffice to say that this line uses the **index** keyword, and supplies four arguments which are JSON strings.

Another directive using the **domain** keyword is:

```
domain "mydomain.com"
```

Directives do not all use JSON strings. Other JSON values are also used:

```
mail_use_ssl                                    true
```

Uses the **mail_use_ssl** keyword, and specifies that when sending email, the application must use SSL.

Similarly:

```
secpol_inactivity_timeout                    600
```

Indicates that the security policy the application will enforce will incorporate a session inactivity timeout of 600 seconds.

Inclusion

The configuration file may be broken down into smaller ones, for one or more reasons This can simply be more practical when a configuration file becomes too large to manage. Smaller ones might also be used to focus on specific aspects of the application. This can promote reuse, with some configuration files reused across various implementations, while other details vary in distinct configuration files that include the common shared files.

A Metaspex application expects all configuration files to be accessible from the single **hx2a.conf** file. The other files can be included through **include** directives. Included files can include other files, of course.

Including another configuration file is done the following way:

```
include "myfile.conf"
```

A configuration file can contain multiple inclusion directives.

Relative and absolute paths are supported; for instance:

```
include "../conf/myfile.conf"
```

Configuration files directives present in included files are interpreted as if they were inserted as is at the point of inclusion. This rule applies transitively to inclusions in included files. Loops are obviously not allowed and are reported if they happen.

Built-in configuration

When the Metaspex application uses the directory, then at the time of the directory creation, Metaspex will automatically create a number of indexes or views (depending on which database management system is used).

Metaspex supports Couchbase Global Secondary Indexes (GSI) and Couchbase Views. It also supports CouchDB Views. We'll see later in this document that indexes and views are specified using the **index** and **view** configuration file keywords.

If you want to use a directory which will be implemented using Couchbase GSI or MongoDB, you will have to include the file **dirindex.conf**, which is delivered with the Metaspex distribution. If you want to use Couchbase or CouchDB Views, you must include **dirview.conf**.

These files make use of standard configuration file keywords.

Tracing

Metaspex offers two ways, which can be used concurrently, to observe what is happening in an application. One is tracing in log files, the other one is the audit database. Let's look first at tracing. (We will come to the audit database in the section below titled "Databases.")

The keyword to use to activate tracing is **log_min_severity**. It can be used, for instance, in this way:

```
log_min_severity "trace"
```

Metaspex supports logging statements in applications. Each statement specifies its level of logging, one of **trace**, **debug**, **info**, **warning**, **error** or **fatal**.

```
HX2A_LOG(trace) << "Hello World!";
```

Is a typical line added in Metaspex code in order to add the line "Hello World!" to the log file when this instruction is encountered. It will be added only if the logging minimum severity is set to "trace" in the configuration file, as above.

The inserter shown above can use local variables to display additional useful information in the log file.

The arguments allowed for the **log_min_severity** keyword are: "trace", "debug", "info" or "warning". Of the logging levels, **error** and **fatal** are always logged, and the default behavior is to log only these two. Lower severities are not logged by default, even if logging statements with these severities are executed in the code. You can modulate the amount of logging from the configuration file without changing the code or even recompiling the application.

Metaspex allows fine control over the naming and rotation of log files. Please consult the reference guide for details about that. Just be aware that log files can produce vast amounts of data on disk and you'll need a housekeeping process to archive or delete older log files.

Variables

A handy feature of Metaspex configuration files is the definition of symbolic names associated with values.

Instead of hardcoding values in the code, you can define them in a configuration file, associate a name with them, and then access the value from the code by giving the name. Any JSON value or UUID can be defined this way.

When you need a different application behavior, there is no need to change the code or even to recompile it. Just change the value in the configuration file that contains it, and restart the application.

Here is an example:

```
number "max_items" 100
```

It uses the **number** keyword. Once this definition has been made in the configuration file, the code can contain the following:

```
config::get_number("max_items")
```

This will return a C++ double with the value 100, which can be used in whatever context you need it.

Metaspex also supports keywords **string** and **boolean**:

```
string "mystring" "Hello World!"
boolean "myboolean" true
```

They can be accessed from code respectively as a C++ string and boolean with:

```
config::get_string("mystring")
config::get_boolean("myboolean")
```

Metaspex supports also UUIDs/document identifiers:

```
id "myuuid" "7e605d1f85834dfdb1b37988e953d1f9"
```

```
config::get_id("myuuid")
```

Will return the corresponding strongly-typed UUID. This is often used in **database** and **index**/**view** definitions, not for dynamic code access but for shortening definitions in the configuration file.

The keyword **json** is also accepted. The second argument is a string containing a valid JSON value.

Databases

A Metaspex application is a high-level set of requirements turned directly into code. As such, it does not contain any database API or any database query language. As a result, the same requirements should be able to run on top of any of the supported databases, even if these databases offer completely different APIs or query languages, and even if the database offers no query language whatsoever (e.g., CouchDB).

Strictly speaking, a very few features are not available across all supported databases, but the differences are not substantial (and they have been noted in the technical insights series when they have come up).

Metaspex goes further than simply allowing the same code to run on a variety of database products: it allows you to run any Metaspex-based binary on any of the supported databases. To run the same binary on another database is just a matter of changing the configuration file. You compile once, you deploy multiple times on different environments.

Of course, a Metaspex application can use several distinct database products at the same time. Unique features such as **link**s can be established across documents sitting in different database products, and referential integrity will operate correctly across these. Your code will not be aware of the existence of different database products.

It means that in the configuration file, you can specify multiple databases, and you have complete freedom to choose any of the supported database products for each of them.

There are three types of databases you can declare in a configuration file:

1. The optional directory. There can be at most one in the configuration file. It uses the **directory** keyword.
2. The audit database. The keyword is **auditdb**.
3. Regular databases. The keyword is **database**.

Let's look at them one by one.

The directory

Here is a typical configuration file **directory** declaration:

```
directory "couchbase://bucketuser:bucketuserpwd@dbhost"
```

The **directory** directive takes a single string. The first part of this, before the colon, is the database product. Here, it is Couchbase. After the double slashes, we find the database user that the application will use to interact with the Couchbase instance storing the directory. After the database user, we find its password. Last, we see the host name of the machine running the Couchbase instance, as defined in the operating system. You can use a remote host name, **localhost**, or an IP address here.

In case the directory was stored in MongoDB, we would have:

```
directory "mongodb://bucketuser:bucketuserpwd@dbhost"
```

And, for CouchDB:

```
directory "couchdb://bucketuser:bucketuserpwd@dbhost"
```

Of course, the password for the database user being confidential, it means that in production, the user should protect the configuration file by means provided by the host operating system, so that it is only readable by the operating system user running the application.

The audit database

The audit database is a way to log every web service interaction with a backend, in a way much more sophisticated than simple log files.

Every session token, every query payload, and every reply is logged in the audit database. The audit database can be used for forensics, and indexes can be created to select specific events between two timestamps.

The audit database is optional. Declaring the audit database in the configuration file is enough for Metaspex to start using it.

An audit database uses the **auditdb** configuration file keyword. It takes one argument, formatted as for the **directory** we just saw.

Regular databases

Regular databases follow the same principle as the **directory**, except that they take a logical name as first argument of the **database** keyword, and the second argument ends with the name of the bucket to be used:

```
id "mydbid" "0acfc7eef68147bab15b55bc8517219f"
database "mydbid" "couchbase://bucketuser:bucketuserpwd@dbhost/mydb"
```

We first define a UUID variable, here named "mydbid", and we associate a UUID, which can be obtained by using the **hx2a uuid** utility command.

We reuse the name "mydbid" as the first argument of the **database** keyword. In the second argument, note the end of the string: "/mydb". This is the name of the bucket to use when interacting with the "mydbid" database. As we saw previously in the technical insight on

making objects, the code will create a database connector using the "mydbid" string. Metaspex will find all the characteristics it needs to interact properly with that bucket.

We'll see later that from the configuration file, a Metaspex utility can take care of creating all the regular databases and the database users, including their passwords, with a single command. The directory and the audit databases are created separately with specific commands. In their case also, the utility will take care of creating them, irrespective of the physical database product details being used. The utility will take care of everything.

Indexes and views

We have a specific technical insight on indexes. It presents the **index** keyword. The **index** keyword can be used with Couchbase to refer to Global Secondary Indexes (GSI) or with MongoDB. When using CouchDB or Couchbase Views, the **view** keyword needs to be used instead.

As a reminder, an **index** declaration looks like:

```
index "hx2a" "mydescidx" "mytype" "a.b:c.d"
```

Similarly, a **view** looks like this:

```
view "hx2a" "mydescidx" "mytype" "a.b:c.d"
```

As with **database**s, a Metaspex utility will create all the **index**es and **view**s specified in the configuration file in one single call. It will adjust to the database products being used to perform the task with the specific database product tools available. In the case of CouchDB, it will generate the JavaScript code to build and maintain the **view**s.

Automated implementation

Metaspex comes with the **hx2a configure** utility. It inspects the configuration file and creates all the regular databases and indexes using the appropriate database product tools made available by the vendors.

No risk of error, no risk of missing something is possible. The **configure** utility is entirely autonomous and it performs a complete implementation.

The only exception is the directory and the audit database. They are created using the **hx2a mkdirectory** and **hx2a mkauditdb** utilities, respectively.

Encryption and hashing keys

Metaspex supports symmetric encryption and cryptographic hash for session tokens. It also supports symmetric encryption for **slot**s.

Naively storing the corresponding keys directly in the code violates security and makes maintenance very difficult. Metaspex accordingly requires the definition of the corresponding keys in the configuration file.

The code is compiled once, and when deployed, it uses the keys stored in the configuration file. Different customers can use the same binaries, but different keys in their configuration file, without running the risk of being hacked by a leak from another customer.

Please consult the reference guide for the specific keywords to use to specify the keys.

Security policy

The Metaspex configuration file supports more than 30 keywords to specify the security policy. They all start with the prefix “**secpol_**”.

We have a dedicated technical insight on security, which provides details.

Configuring sending mail, SMS and in-app notifications

Metaspex offers abstractions to send mail, SMS and make in-app notifications. These facilities are controlled by an extensive set of configuration data.

The Metaspex configuration file allows you to specify everything which is necessary, making the code independent from all these credentials and parameters. See the reference guide for details.

Conclusion

Metaspex adopts the stance: “Code and compile once, deploy many times and many ways.”

The benefit of this is that the requirements, under the form of code, are devoid of any implementation details—including confidential ones like keys and passwords.

Not only does this offer a “no hassle” way to perform implementations, it also separates building the logical part of applications from their particular deployment details.

The behavior of a given application can vary greatly from one implementation to another, and even with a single implementation, the evolution of the application can be finely directed from the keyword-rich configuration offered by Metaspex.

Applications are easier to specify and they stay safe even in the face of evolving implementation and deployment requirements.