NCSA Security Library Configuration Framework

Overview

Various systems need a configuration file. NCSA supports a very specific Configuration Framework aka **CF**, (basic system you extend to roll your own). The basic gist is a set of configurations that have a name. You may then retrieve your configuration by name. The framework allows for:

- Importing other configurations – point to other configurations and they will be seamlessly imported
- Setting environment variables – You may set environment variables and reference them in your configuration, allowing great flexibilty.
- Aliases – You may create your custom configurations and, rather than having a body, set an alias to point to another configuration. This lets you, e.g., just specify loading a configuration name ***default*** but have the specific configuration possibly change. You service or application then can be configured once and the configuration itself can change to reflect your requirements.
- Inheritance – you may have configurations inherit from other configurations. Multiple inheritance is supported.

Basic structure

The basic structure for an XML document is

```
<bundle>
     <file include="path"/>*
     <env key | include>[value]</env>*
     <custom name {alias} extends=list></custom>
</bundle>
```

The bundle element

This is the outermost wrapper for the framework. Every element the system recognizes is in this element. When we say ***bundle*** we refer to a collection, set or whatever you want to call it, of configurations, each of which is named uniquely.

Nota Bene: Earlier versions of this system used the tag **`config`** not **`bundle`**. Both are supported though **`bundle`** is preferred.

The file element

The file element has a single attribute, the fully qualified path to a file. Note that the file is another CF file. Files may load files. Note that circular references will be caught and an exception thrown, so A loads B which loads A would be caught.

The env element

The environment is simply a set of keys and values. The keys are accessed in the configuration using the syntax **${key}** You may have these *anywhere* in the file (including in env elements and file elements). Your configuration may even have them in XML tags so that, until resolution, it is not a valid XML document. All environment tags are pre-processed and resolved before the main configuration is loaded.

Scope of environment variables

The environment element resides in the top level of the configuration. You may either insert environment elements anywhere in this, but they will be extracted and processed together. Variables are scoped to that file and files it loads. So if a bundle, A, sets environment variables and includes a bundle, B, then all of the environment variables in A are also available in B. B may override them too. However, variables loaded in B stay scopes there and are not available in A. This mimics standard scoping for functions, for instance.

custom elements.

To actually use the framework, you create a single configuration in XML. This is then given a tag by you (labelled *custom* above). These all reside in the top level. When you load your configuration file, you specify what the custom tag is and the system then applies the framework to it, handing you back the **`CFNode`** object that represents it. You may either navigate this directly or use various utility methods in that class. The aim is once you specify your own configuration, the headache of slogging through XML tags is done for you.

An Example

A very simple example of using this framework. You have a custom tag of **server**, In this, you have two configurations, one named **default** which just points to the current configuration you want, and the other named **mysql.8.3**. Note that you could have several such named configurations if you had to support different MySQL versions and just choose the right one. This is extremely useful when migrating to another version since you can back your server out of any changes with a minor configuration change.

There are two environment variables defined, **root_dir** which points to your files on disk and **server** which is the address your service should use (in this case for MySQL).

```
<bundle>
   <env key="root_dir">/path/to/files</env>
   <env key="server">localhost:4452</env>

   <server name="default" alias="mysql.8.3"/>
```

```
   <server name="mysql.8.3">
       <keys jwk_file="${root_dir}/etc/keys.jwk"/>
       <mysql address="jdbc:mysql://${server}/my_service"/>
   </server>
</bundle>
```

Effective post-proccessed configuration named **default**:

```
   <server name="default">
       <keys jwk_file=">/path/to/files/etc/keys.jwk"/>
       <mysql address="jdbc:mysql://localhost:4452/my_service"/>
   </server>
```

i.e., this is what your application "sees".

Using the system

The maven dependency

You specify the dependency via maven

```
<dependency>
    <groupId>edu.uiuc.ncsa.security</groupId>
    <artifactId>storage</artifactId>
    <version>latest</version>
</dependency>
```

Loading a configuration from a bundle

The basic steps are

1. Load the bundle. You may use a file, resource or just create your own stream
2. Get the named configuration you want
3. Perform operations on said configuration to get sub-nodes, attributes etc.

Example

We will show two ways to load the **default** named configuration from the bundle **my_config.xml.**

Using the builder pattern

```
CFBundle bundle = new CFLoader.Builder()
      .tagname("service")
      .inputStream(getFileInputStream("/path/to/my_config.xml"))
      .build()
      .loadBundle();
CFNode service = bundle.cfg.getNamedConfig("default");
```

Using standard Java class calls

```
FileInputStream fis = getFileInputStream("/path/to/my_config.xml");
    CFLoader config = new CFLoader();
    CFBundle bundle = config.loadBundle(fis, "service");
     CFNode service = bundle.cfg.getNamedConfig("default");
```

Both do the same thing.

Using the configuration.

Like XML, the configuration is logically divided into nodes. The CFNode class allows you to navigate your configuration with convenient calls, handling all of the messy details. The major calls are

- getFirstAttribute(attributeName) – returns the first attribute value (as a string) from the current node.
- getAttributes(attributeName) – returns all of the attributes for the given name
- getFirstNode(nodeName) – returns the CfNode child with the given name
- getNodes(nodeName) – returns a list of all the CFNodes that are children of this node.

There are other convenience methods too, such as getFirstBooleanAttribute which returns the first attribute, possibly with a default. Note that boolean values are **true|on** or **false|off** and case insensitive. If you use one of the filters for attributes, then only such attributes are returned and attributes that are not of the type are skipped.

Example

Let us say you had the following named configuration in your bundle:

```
  <sas name="production"
       enabled="true">
     <fileStore path="/opt/oa4mp/var/storage/file_stores/sas">
        <clients indexOn="true"/>
     </fileStore>

     <JSONWebKey defaultKeyID="2D700DF531E09B455B9E74D018F9A133">
        <path><![CDATA[/opt/oa4mp/etc/sas-keys.jwk]]></path>
     </JSONWebKey>
  </sas>
```

Then you could navigate it as

```
CFNode sasConfig= bundle.cfg.getNamedConfig("production");

// Check that the configuration is enabled.
if(!sasConfig.getFirstAttribute("enabled"){ // do something }


// Load the fileStore (assuming load() takes the file path and
// returns the store you want.
CFNode fsConfig = sasConfig.getNode("fileStore");
FileStore fileStore = load(fsConfig.getFirstAttribute("path"));

// Check that the
if(fsConfig.getFirstNode("clients").getFirstBooleanAttribute("indexOn")){
  // then do something with an index
}

// And so forth to get the JSONWebkey
```

Again, part of the CF is to hide grisly complexities of XML from you and allow you to manage possibly very many configurations easily, choosing them as needed.