An Overview of

Config4☆

www.config4star.org

Copyright

What is Config4*?

- Config4* is pronounced “config for star”:
  - The pedantically correct “config for asterisk” does not sound as good
  - The “*” is a placeholder for the name of a programming language

- Config4* is a configuration-file parser for several languages:
  - Config4Cpp (C++ version)
  - Config4J (Java version)
  - Other programming languages may be added in the future

- Config4* is open-source and uses the MIT license:
  - Compatible with most/all other open-source and proprietary licenses

- The project website is www.config4star.org
  - Provides source code and comprehensive documentation

Why is Config4* so good?

- Config4* has useful features rarely found in other configuration technologies:
  - Each feature is useful in its own right
  - In addition, there is synergy in the interaction of the features
- Structure of this presentation:
  - First, explain the basic features of Config4*
    (similar to features in competing technologies)
  - Then, discuss the useful features, and the synergies between them

1. Basic features

Basic features of syntax

```
# this is a comment

@include "another_file.cfg";

string_variable = "value";
list_variable = ["a", "list of", "values"];

string_concatenation = string_variable + "...";
list_concatenation = list_variable
                   + ["another", "list"];

scope_name {
    variable_inside_scope = "value";
    nested_scope {
        yet_another_variable = "...";
    }
}
# scoping operator is "."
# Example: "scope_name.variable_inside_scope"
```

Keywords are prefixed with "@" to prevent clashes with names of variables or scopes

Basic API (Java syntax)

```
import org.config4j.*;
...
Configuration  cfg        = Configuration.create();
String         configFile = "file.cfg";
String         scope      = "foo";
String[]       fontList;
String         logFile;
int            logLevel;
try {
    cfg.parse(configFile);
    logFile  = cfg.lookupString(scope, "log_file");
    fontList = cfg.lookupList(scope, "font_list");
    logLevel = cfg.lookupInt(scope, "log_level")
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

Notes

- A lookup operation merges its `scope` and `localName` parameters to form a fully-scoped name:
  - Example, `lookupString("foo", "bar")` → `"foo.bar"`
  - The `scope` parameter is usually set from a command-line argument
  - One configuration file can contain scopes for many applications
- Data-type conversion:
  - Some lookup operations call `lookupString()` and then convert to the desired type
  - Examples: `lookupInt()`, `lookupFloat`, `lookupBoolean()`, …
- An existing configuration-file parser:
  - Might not have *all* the features shown on the previous two slides
  - But such features are sort-of common
- Now let's look at useful Config4* features rarely found elsewhere…

2. Centralised configuration

Description of a common problem

- The Acme company makes and sells software:
  - Small customers will run the software on one computer
  - These customers want the convenience of a configuration file
  - Large customers will run the software on hundreds of computers
  - They do not want to maintain hundreds of copies of a configuration file
  - They insist on having a centralised configuration repository
- It might cost Acme a lot of time and money to implement a centralised configuration mechanism:
  - Complexity of a client-server architecture?
  - Use a database? Administration skills required. License costs?
  - Extra complexity and expense if fault tolerance is required
- Config4*, with help from a utility called Curl, provides a zero-cost solution

The synergy of Config4* and Curl

- Curl (an abbreviation of “Client for URL”):
  - Is an open-source utility, available for most operating systems
  - Retrieves a file from a specified URL and echoes it to the console
  - Supports many protocols: HTTP, FTP, LDAP, ...
- Example: `curl -sS http://www.example.com/file.cfg`
  - The “`-sS`” option instructs curl to not print any diagnostics
- Config4* can parse:
  - A file: `cfg.parse(“file.cfg”)`
  - The output of executing a command:
    `cfg.parse(“`**`exec#`**`curl -sS http://...”)`

    Command to execute

The synergy of Config4* and curl (cont')

- Benefit for Acme. It satisfies small and large customers:
  - Small customers use a "`file.cfg`" command-line argument
  - A large customer runs software on many computers, and specifies "`exec#curl -sS http://centralisedHost/file.cfg`" as a command-line argument
- Benefits for large customers:
  - They can use any protocol supported by `curl`
  - They are not restricted to using only `curl`
    (they can use a utility that retrieves configuration from, say, a database)
  - They can use a fault-tolerant database or web server, if needed
  - As more and more applications use Config4*, the cost of maintaining a centralised database or web server is amortised

3. Fallback configuration

The goal of “install and use” for applications

- “Install and use” applications are convenient to use:
  - Similar to “plug and play” hardware
  - Unfortunately, many applications requires a configuration step before use
  - The need to “configure before use” can irritate users

- Ideally, an application will have an *optional* configuration file:
  - Embedded configuration means it can run *without* a configuration file (thus bypassing the need to “configure before use”)
  - The embedded configuration can be overridden with an external configuration file

- Config4* enables developers to achieve this goal in two steps:
  - Run the `config2cpp` or `config2j` utility to create embedded configuration data
  - Call the `setFallbackConfiguration()` operation

The `config2cpp` and `config2j` utilities

- The `config2cpp` and `config2j` utilities:
  - Read a configuration file, and
  - Generate a C++ or Java class that provides a snapshot of the file's contents
- This provides “fallback” configuration data that can be embedded in an application
- Examples of use:

```
config2cpp -cfg Fallback.cfg -class FallbackConfig
           -singleton
config2j -cfg Fallback.cfg -class FallbackConfig
           -singleton
```

- The generated class provides a `getString()` operation that returns the “fallback” configuration data

The `setFallbackConfiguration()` operation

- An application sets fallback configuration as follows:

```
Configuration cfg = Configuration.create();
String cfgFile = ...;
try {
    if (cfgFile != null) { cfg.parse(cfgFile); }
    cfg.setFallbackConfiguration(
                        Configuration.INPUT_STRING,
                        FallbackConfig.getString());
    logFile = cfg.lookupString(scope, "log_file");
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

- The Config4* lookup operations work as follows:
  - Search for the specified variable in the configuration object
  - If found, then return its value
  - Otherwise, search for the specified variable in the fallback configuration

Synergy of fallback and centralised configuration

- Fallback and centralised configuration are independent features
  - But they interact to provide synergy

- A Config4*-based application:
  - Can use fallback configuration to provide “install and use” convenience for new users
  - Can use an external configuration file to override fallback configuration
  - Can use “`exec#curl ...`” if the user deploys the application on many computers

- Thus:
  - Config4* scales from single-user to enterprise deployments (you could even use fallback configuration in an embedded system)
  - It is difficult to think of another configuration technology that provides this level of flexibility

4. Adaptable configuration

Description of a common problem

- Often, the contents of a configuration file change when:
  - Moving the application from one computer to another
  - Running the application under another user name
- Editing a configuration file to make such changes is tedious
- It would be better if a configuration file could automatically adapt to its runtime environment:
  - Then, the *same* configuration file could be used on multiple computers and by multiple users
  - Config4* provides excellent support for this

The `getenv()` and `exec()` functions

- The `getenv()` function:
  - Returns the value of an environment variable
  - Is typically used to access the name of the user or the installation directory for software: `getenv("USERNAME"), getenv("FOO_HOME")`

- The `exec()` function:
  - Executes a command and returns its standard output
  - Is typically used to determine the host name: `exec("hostname")`
  - A security mechanism prevents execution of malicious commands

- These operations, combined with the "+" operator, enable a configuration file to adapt to its runtime environment. Example:

```
log_dir = getenv("FOO_HOME") + "/logs/"
                             + exec("hostname");
```

If-then-else statements and `osType()`

```
production_hosts = [“pizza”, “pasta”, “cheese”];
test_hosts = [“foo”, “bar”, “widget”, “acme”];

@if (exec(“hostname”) @in production_hosts) {
    server_x.port = “5001”;
    server_y.port = “5002”;
    server_z.port = “5003”;
} @elseIf (exec(“hostname”) @in test_hosts) {
    server_x.port = “6001”;
    server_y.port = “6002”;
    server_z.port = “6003”;
} @else {
    @error “This is not a production or test machine!”;
}

@if (osType() == “windows”) {
    ...
} @else { # UNIX
    ...
}
```

Adapting to command-line options

- Users may want to use command-line options to override variables in a configuration file
- Config4* supports a two-step approach for doing this:
  - Before parsing a configuration file, the application calls `insertString()` to insert *name-value* pairs obtained from the command line
  - The configuration file uses the conditional assignment (`"?="`) operator to provide default values for variables
- The following slides illustrate these steps

Using the `insertString()` operation

```
import org.config4j.*;
...
Configuration  cfg = Configuration.create();
for (int i = 0; i < args.length; i++) {
    if (args[i].equals("-set")) {
        cfg.insertString(scope, args[i+1], args[i+2]);
        i = i + 2;
    }
}
try {
    cfg.parse(configFile);
    ... // calls to cfg.lookup<Type>() operations
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

Using the conditional assignment ("?=") operator

- The `?=` operator assigns a value to a variable *only if* the variable does not already exist
- Example of syntax:

```
log_level ?= "0";
username  ?= getenv("USERNAME");
password  ?= "";
```

- Typically, such variables can be pre-set by command-line options that are processed as shown on the previous slide
- In this way, a configuration file can adapt to (be overridden by) command-line options

Synergy

- Adaptable configuration is independent of centralised configuration
- However, those features can interact to provide synergy.
- Example:
  - A large company deploys an application on 500 computers
  - A single configuration file is stored on a web server and accessed via "`exec#curl ...`"
  - That centralised configuration file can use `@if`-then-`@else`, `getenv()` and `exec()` to adapt its contents for each computer

5. Useful data types

Durations

- Some configuration files need to specify durations:
  - For example, connection timeout, idle timeout, transaction timeout…
  - In most configuration files, these are expressed as integer values (denoting seconds or milliseconds)

- Example from a product that does *not* use Config4*:

```
refresh: 28800
retry:   7200
expire:  1209600
minimum: 86400
```

Expressed in seconds

- Equivalent in Config4* syntax:

```
refresh = "8 hours";
retry   = "2 hours";
expire  = "2 weeks";
minimum = "1 day";
```

Units can also be `minutes`, `seconds` or `milliseconds`

Durations (cont')

- Config4* can convert durations into seconds or milliseconds:

```
x = cfg.lookupDurationSeconds(scope, "refresh");
y = cfg.lookupDurationMilliseconds(scope, "retry");
```

- The value `"infinite"` is converted into the value `-1`

Memory sizes

- Config4* also supports memory sizes
- Examples of syntax:

```
buffer_size  = "512 bytes";
cache_size   = "32 KB";
max_log_size = "1.5 GB";
```

- Config4* can convert memory sizes into bytes, KB or MB:

```
lookupMemorySizeBytes(scope, localName)
lookupMemorySizeKB(scope, localName)
lookupMemorySizeMB(scope, localName)
```

Other data-type conversions

- Other operations enable you to quickly write code to:
  - Convert string values to integer constants (like `enum` in C/C++)
  - Example: `"red"` → `0`, `"green"` → `1`, `"blue"` → `2`
  - Parse values `"<units> <number>"` or `"<number> <units>"`
  - Examples:
    - `"£19.99"` → (`"£"`, `19.99`)
    - `"42 cm"` → (`"cm"`, `42`)
- You can also process a list as if it were a table. Example:

```
price_list = [
 # description    unit price
 #---------------------------
   "apple",       "£0.49",
   "orange",      "€2.99",
];
```

6. Schema validation

Benefits of schema validation

- A *schema* is a blueprint or definition of a system.
- Examples:
  - A database schema defines the layout of a database
  - DTD, XML Schema and RELAX NG are competing schema languages for defining the permitted contents of an XML file
- Config4* has a schema language, which provides:
  - An intuitive, easy-to-use syntax
  - An easy-to-use API
  - The ability for developers to define new schema data-types.

Example configuration scope

- An application uses a configuration scope like that shown below:

```
foo_server {
    idle_timeout = "2 minutes";
    log_level = "3";
    log_file = "/tmp/foo.log";
    price_list = [
      # time       colour    price
      #-----------------------------------
        "shirt",  "green",  "EUR 19.99",
        "jeans",  "blue",   "USD 39.99",
    ];
}
```

- The next slide shows how to perform schema validation for such a scope

Example of schema validation (Java syntax)

```
Configuration cfg = Configuration.create();
SchemaValidator sv = new SchemaValidator();
String schema = new String[] {
    "@typedef colour = enum[red, green, blue]",
    "@typedef money = units_with_float[EUR, GBP, USD]",
    "idle_timeout = durationMilliseconds",
    "log_level = int[0,5]",
    "log_file = string",
    "price_list = table[string,item, colour,colour,
money,price]"
};
try {
    cfg.parse(configFile);
    sv.parseSchema(schema);
    sv.validate(cfg, "foo_server", "");
} catch(ConfigurationException ex) {
    System.err.println(ex.getMessage());
}
```

- A descriptive exception is thrown if schema validation fails

Comparison with XML Schema

- XML Schema:
  - Is very verbose
  - Has a steep learning curve:
    - Syntax specification is written in impenetrable legalese (about 380 pages long if you convert it from HTML into PDF format)
    - Good books on XML Schema are 400–500 pages long
  - Provides difficult-to-understand error messages

- In contrast, the Config4* schema language:
  - Is very concise
  - Is intuitive and easy to learn:
    - Syntax specification, with examples, is defined in 13 pages
  - Provides easy-to-understand error messages

7. Reuse with the `@copyFrom` statement

Description of problem

- Some applications are related to other applications. Examples:
  - Applications that are developed as part of the same project
  - Replicas for a server application
- Such applications may have similar configuration settings:
  - Most variables have identical values
  - A few variables have different values
- Can such applications reuse the variables with identical values?
  - Doing this can significantly reduce the size of configuration files
  - The `@copyFrom` statement facilitates this

Example of the `@copyFrom` statement

```
server.defaults {
    timeout = “2 minutes”;
    log {
        dir = getenv(“FOO_HOME”) + “/logs”;
        level = “2”;
    }
}

foo_server {
    @copyFrom “server.defaults”;
    log.level = “1”; # override copied value
}

bar_server {
    @copyFrom “server.defaults”;
    timeout = “30 seconds”; # override copied value
}
```

Conditional `@include` and `@copyFrom`

- Config4* provides conditional variations of the `@include` and `@copyFrom` statements
- These help a configuration file adapt to its environment. Examples:

```
@include getenv("HOME") + "/.foo.cfg" @ifExists;

override.pizza { ... }
override.pasta { ... }
foo_server {
  ... # set default values
  @copyFrom "override." + exec("hostname") @ifExists;
}
```

8. The “`uid-`” prefix

The "uid-" prefix

- Let's assume you want to store details about employees
- You might try the following:

```
employee { name = "John Smith"; ... }
employee { name = "Jane Doe"; ... }
```

- That will not work:
  - Because the second occurrence of `employee` re-opens the *existing* scope, so the details of Jane Doe override those of John Smith
- Config4* provides a "`uid-`" prefix for such situations:
  - "uid" is an abbreviation for "unique identifier".
  - Config4* expands each name of the form form `uid-employee` into `uid-<unique-number>-employee`

The "uid-" prefix (cont')

- Fixed example:

```
uid-employee { name = "John Smith; ... }
uid-employee { name = "Jane Doe"; ... }
```

- The Config4* API provides operations for processing uid entries
- The "uid-" prefix makes Config4* suitable for use as a data-file format

9. Comprehensive documentation

Comprehensive documentation

- Many open-source software projects provide minimal or no documentation:
  - Lack of documentation creates a frustrating learning curve for users
  - You might have to spend days or weeks “playing with” a project to learn if it suits your needs
- In contrast, Config4* provides comprehensive documentation:
  - Getting Started guide
  - C++ and Java API guide
  - Practical Usage guide (this provides expert advice)
  - Maintenance guide (for people interesting in extending/porting Config4*)
- The high-quality documentation (400 pages in total):
  - Significantly reduces the learning curve
  - Enables you to quickly decide if Config4* suits your needs

10. Summary

Summary

- Config4* provides features common to many other configuration technologies

- Config4* *also* provides:
  - Fallback (embedded) configuration for “install and run” applications
  - Centralised configuration (vital for large deployments)
  - Adaptable configuration:
    - `@if`-then-`@else`, `getenv()`, `exec()`, `osType()`
    - The `?=` operator enables integration with command-line options
  - Reusable configuration (the `@copyFrom` statement)
  - Scopes enable one file to store configuration for multiple applications
  - Useful data-types: duration, memory sizes, tables, …
  - Schema validation
  - The “`uid-`” prefix (can use Config4* as a data-file format)
  - Comprehensive, high-quality documentation