DAVID BLANK-EDELMAN

practical Perl tools

CONFIGURATION FILES

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of *Perl for System Administration* (O'Reilly). He has spent the past 20 years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the chair of the LISA 2005 conference.

dnb@ccs.neu.edu

LET ME PUT DOWN THE "UNDER NEW management" sign for a moment and welcome you to the first article in a slightly different column than you are used to seeing in this spot. It would be very difficult to step into Adam Turoff's shoes, especially given his multiple years of great articles, so I'll be taking this column in a different direction.

It is most auspicious that the theme of this issue of *;login:* is "system administration" because that's my particular bent as well (having proudly been in the biz for about 20 years). As a sysadmin I've been interested in Perl for many of those years because it has been a good tool for lots of practical uses: hence the new column title. (To get the heresy out of the way early in my tenure: I don't think Perl is always the best tool. You should always use the best tool for the job.) So that's enough meta-yammering about the column; let's get on to the actual subject of today's article.

Let us consider the lowly config file. For better or worse, config files are omnipresent not just for a sysadmin but for anyone who has ever had to configure software before using it. Yes, GUI and Web-based point-and-click festivals are becoming more prevalent for configuration, but even in those cases there's often some piece of configuration information somewhere in a file that has to be twiddled before you can even get to that point in the setup of new software.

From the Perl programmer's point of view (ours), the evolutionary stages of a program usually go as follows.

First, the roughest, simplest of scripts (this stage may be skipped by senior programmers):

```
use strict; # assume this line for all of our examples

open my $DATA_FILE_H, '<', "/var/adm/data"
    or die "unable to open datafile: $!\n";
open my $OUTPUT_FILE_H, '>', "/var/adm/output"
    or die "unable to write to outputfile: $!\n";

while ( my $dataline = <$DATA_FILE_H> ) {
    chomp($dataline);
    if ( $dataline =~ /hostname: / ) {
        $dataline .= ".example.edu";
    }
    print {$OUTPUT_FILE} $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;
```

That's quickly replaced by the next stage, the arrival of variables:

```
my $datafile       = '/var/adm/data';       # input data file name
my $outputfile     = '/var/adm/output';     # output data file name
my $change_tag = 'hostname: ';              # append data to these lines
my $fdqn           = '.example.edu';        # domain we' ll be appending

open my $DATA_FILE_H, '<', $datafile
    or die "unable to open $datafile: $!\n";
open my $OUTPUT_FILE_H, '>', $outputfile
    or die "unable to write to $outputfile: $!\n";

while ( my $dataline = <$DATA_FILE_H> ) {
    chomp($dataline);
    if ( $dataline =~ /$change_tag/ ) {
        $dataline .= $fdqn;
    }
    print {$OUTPUT_FILE} $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;
```

Many Perl programs happily remain at this stage for the duration of their lifespan. However, more experienced Perl programmers recognize that code like this is fraught with potential peril when development continues and the program gets bigger and bigger, perhaps being handed off to other people to maintain. This peril manifests the first time someone naïvely adds code deep within the program that modifies $change_tag or $fdqn. All of a sudden the output of the program changes in an unexpected and unwanted way. In a small code snippet it is easy to spot the connection between $change_tag or $fdqn and the desired results, but it can be much trickier to find something like this in a program that scrolls by for many screensful.

One approach to fixing this problem would be to rename variables like $fdqn to something more obscure such as $dont_change_this_value_yesiree_bob, but that's a bad idea. Besides consuming far too many of the finite number of keystrokes you are going to be able to type in your lifetime, it wreaks havoc on code readability. There are a number of data-hiding tricks we could play instead (closures, symbol table manipulation, etc.), but they don't help with readability either and are more complex than is necessary. The best idea is to use something similar to the "use constants" pragma to make the variables read-only:[1]

1. Why not actually "use constants" instead? The Readonly module documentation points out a number of reasons. The three most compelling are: the ability to interpolate Readonly variables into strings (e.g., print "Constant set to $CONSTANT\n"); the ability to lexically scope the read-only variable (e.g., Readonly my $constant => "fred") so they can be present in only the scope you desire; and unlike "use constant," attempts to redefine a Readonly variable are rebuffed.

```
use Readonly;

# we've uppercased the constants so they stick out
# note: this is the Perl 5.8.x syntax; see the Readonly docs for using
#       Readonly with versions of Perl older than 5.8
Readonly my $DATAFILE          => '/var/adm/data';      # input data file name
Readonly my $OUTPUTFILE       => '/var/adm/output';    # output data file name
Readonly my $CHANGE_TAG    => 'hostname: ';            # append data to these lines
Readonly my $FDQN                => '.example.edu';       # domain we'll be appending

open my $DATA_FILE_H, '<', $DATAFILE
    or die "unable to open $DATAFILE: $!\n";
open my $OUTPUT_FILE_H, '>', $OUTPUTFILE
    or die "unable to write to $OUTPUTFILE: $!\n";

while ( my $dataline = <$DATA_FILE_H> ) {
    chomp($dataline);
    if ( $dataline =~ /$CHANGE_TAG/ ) {
        $dataline .= $FDQN;
    }
    print {$OUTPUT_FILE} $dataline . "\n";
}
close $DATA_FILE_H;
close $OUTPUT_FILE_H;
```

Now that we've seen the ne plus ultra of storing configuration information within the script,[2] we've hit a wall: what happens when we decide to write a second or third script that needs similar configuration information? Any readers who reached for the cut/copy function in their editor as an answer to that question are fired. Simple duplication of the same information into a second script may seem harmless, but it is the first step on to the road away from Oz and toward an unpleasant encounter with the flying monkeys and an unhappy lady with a broomstick. Don't do it.

2. There are some games we could play with the __DATA__ token, but, in general, keeping the configuration information at the beginning of the script is better form.

The right answer may well be some sort of config file.[3] Once you've decided to use a config file, the next question is, What format?

3. If your thoughts sped ahead to more sophisticated solutions, hold on—we'll mention them at the end of this article.

Answering that question is similar to the old joke "The wonderful thing about standards is there are so many to choose from!" Discussions of which formats are best usually become some mishmash of religion, politics, and personal aesthetic taste. Because I'm a flaming pluralist, we're going to take a look at how to deal with several of the most common formats and leave you to choose the best one for your application. I'll try to give you my humble opinion about each to help with that process.

Config File Formats

BINARY

The first kind of configuration file we're going to look at is my least favorite, so let's get it out of the way quickly. Some people choose to store their configuration data on disk as basically a serialized memory dump of their Perl data structures. There are several ways to write this data structure to disk, including the old warhorse Storable:

```
use Storable;

# write the config file data structure out to $CONFIG_FILE
store \%config, $CONFIG_FILE; # use nstore() for platform-independent file

# later (perhaps in another program), read it back in for use
my $config = retrieve($CONFIG_FILE);
```

I've also become fond of the module DBM::Deep, which has the benefit of producing data files that aren't platform-specific by default (though Storable's nstore method can help with that). For a pure Perl module, it is pretty spiffy.

```
use DBM::Deep;

my $configdb = new DBM::Deep "config.db";

# store some host config info to that db
$configdb->{hosts} = {
    'agatha'        => '192.168.0.4',
    'gilgamesh'     => '192.168.0.5',
    'tarsus'        => '192.168.0.6',
};

# (later) retrieve the name of the hosts we've stored
print join( " ", keys %{ $configdb->{hosts} } ) . "\n";
```

Files in this format are typically really fast to read, which can be quite helpful if performance is a concern. Similarly, there's something elegant about having the information stay close to the native format (i.e., a Perl data structure you're going to traverse in memory) for its entire lifespan versus transcoding it back and forth from another representation through a myriad of parsing/slicing/dicing steps.

So why is this my least favorite kind of config file? First, and least palatable to me, is the binary nature of the files created. I'd much prefer to have my config files human-readable wherever possible. I don't want to have to rely on a special program to decode the information (or to encode it, when the data gets written). Besides the visceral reaction, it also means I can't operate on the data using other standard tools such as grep. Luckily, if you are looking for speed, there are other alternatives we'll be discussing in a moment.

NAKED DELIMITED DATA

Also in the category of formats I tend to dislike are those that are simply a set of data in fields delimited by some character. The /etc directory on a UNIX box is lousy with them: passwd, group, and so on. Comma or Character Separated Value files (CSV, take your pick of expansions) are in the same category.

Reading them in Perl is pretty easy because of the built-in split() operator:

```
use Readonly;

Readonly my $DELIMITER   => ':';
Readonly my $NUMFIELDS =>  4 ;

# open and read in a line from your config file here

# now parse the data
my ( $field1, $field2, $field3, $field4, $excess ) =
    split $DELIMITER, $line_of_config, $NUMFIELDS;
```

For CSV files, there are a number of helpful modules to handle tricky situations like escaped characters (i.e., using commas in the data itself, not anything from a prison break). Text::CSV::Simple, a wrapper around Text::CSV_XS, works well:

```
use Text::CSV::Simple;

my $csv_parser = Text::CSV::Simple->new;

# @data will then contain a list of lists, one entry per line of file
my @data = $csv_parser->read_file($datafile);
```

This data format is also on my "least-favored" list. Unlike the previous format, it has the benefit of being human-readable and standard tool-parseable. However, it also has the drawback of being easily human-misunderstandable and mangleable. Without a good memory or external documentation, it is often impossible to understand the contents of the file ("What was the 7th field again?"). This leaves it susceptible to fumble-fingering and subtle typos. It is field-order fragile.

KEY/VALUE PAIRS

The most common format around is the "key {something} value" style, where {something} is usually whitespace, a colon, or an equals sign. Besides the separator difference, there are often other twists like .ini "[sections]" names or configuration scopes (à la Apache's configuration file).

4. For more information on this, Barry Schwartz's book *The Paradox of Choice: Why More Is Less* is recommended.

Dealing with formats like this using Perl modules turns out to be initially hard because there are too many choices.[4] No, really! In my survey of CPAN for config modules for this article, I encountered at least 26 modules of interest that fall into this category. To winnow this down, there are a number of decision forks:

1. How complex do you want the configuration file to be: Will simple .ini files work for you? More complex .ini files? Apache style? Extended Apache style? Do you need sections? Do you need scoped directives? Want to write your own grammar representing the format?
2. How would you like to interact with the configuration information: Want to be handed back a simple data structure? an object representing the information? Prefer to treat things like magical tied hashes or Perl constants? Does the information you get back have to come back in the same order as it is listed in the config file? Would you be happy if the module figured out the config file format for you?
3. What else is important to you: Do you care how quickly the configuration is parsed or how much memory the parsing process takes? Should it handle caching of the config for fast reload? Do you want to be able to cascade the configs (i.e., have a global config with other configs for more specific information)? Should the config be validated on parse?

The answer to each of these questions will point at a different module or set of modules available for your use. We can't dive into all of the modules out there, so let's look at three you may not have seen:

Config::Std is Damian Conway's config parsing module. He's a smart guy and so his module in this space attempts to be the same. Unlike most configuration modules, his module lets you read and then update the configuration file while preserving the section order and the comments. The file format it uses looks much like .ini files, so it should be pretty easy for most people to understand on first sight. Here's an example of the module in action. Note: The examples in this section will be really boring because the modules are all designed to make the process of dealing with config files simple (boring).

```
use Config::Std;
read_config 'config.cfg' => my %config;
# now work with $config{Section}{key}...
# and write the config file back out again
write_config %config;
```

In Conway's book *Perl Best Practices*, he suggests that if you need something more sophisticated than his simple Config::Std format can provide, Config::General can oblige. It handles files in the Apache config file family and has a much richer syntax. Actual use of the module isn't any more complex than Config::Std:

```
use Config::General;
my %config = ParseConfig( -ConfigFile => 'rcfile' );
# now work with the contents of %config...
# and write the config file back out again
SaveConfig( 'configdb', \%config );
```

Config::Scoped gives you still more bells and whistles. It parses a similarly complex format that includes scoped directives (essentially the one used by BIND or the ISC DHCP server), can check the data being parsed, will check the permissions of the config file itself, and includes caching functionality. This caching functionality allows your program to parse the more complex format once and then quickly load in a binary representation of the format on subsequent loads if the original file hasn't changed. This gives us the speed we coveted from the first kind of file we looked at and the readability of the file formats discussed in this section. It doesn't, however, offer an easy way to programmatically update an existing configuration file like some of the other modules we've seen. Here's a small snippet for how to use the caching functionality:

```
use Config::Scoped;
my $parser= Config::Scoped->new( file => 'config.cfg' );
my $config= $parser->parse;

# store the cached version on disk for later use
$parser->store_cache( cache => 'config.cfg.cache' );

# (later, in another program...)
$cfg = Config::Scoped->new( file => 'foo.cfg' )->retrieve_cache;
```

If you are the type of person who likes to smelt your own bits, then there are also a number of other modules like Config::Grammar which allow you to define your own grammar to represent the configuration file format. I tend not to like creating custom formats, if I can help it, for reasons of maintainability, but if this suits your purposes these modules can oblige.

MARKUP LANGUAGES

The last format type we'll be looking at is becoming increasingly common as XML continues to pervade more and more of the IT space (largely due to its shininess). And the use of a markup language like XML to describe configuration information is becoming more and more prevalent.[5] Marketing potential aside, XML does have a few nice properties when used for config files. When kept simple (because a complex/convoluted XML document is as inscrutable as one in any other format), XML config files can be nearly self-documenting. The freedom to define almost arbitrary tags lets it be as descriptive as you'd like. If I write a simple XML file like this, you can probably understand the gist of it without needing a separate manual page:

5. There are in fact XML dialects such as DCML, NetML, and SAML which are gunning for parts of the configuration management space.

```
<config>
   <host>
      <name> agatha </name>
      <addr> 192.168.0.4 </addr>
   </host>
   ...
</config>
```

Another plus of this format is the well-defined syntax and optional validation mechanisms which are part and parcel of XML. At the very least, this means that all of your XML config files can share the same parser and validation mechanism independent of their actual content.

The easiest way to read an XML config file from Perl is the XML::Simple module. It allows you to write simple code like this to slurp an XML file into Perl data structure:

```
use XML::Simple;

my $config = XMLin('config.xml');

# work with $config->{stuff}
```

Turning that data structure back into XML for writing after you've made a change to it is just as easy:

```
... (data structure already in place)
open my $CONFIG_FILE_H, '>', $configfile
    or die "Can't write to $configfile:$!\n";
print {$CONFIG_FILE_H} XMLout($config);
close $CONFIG_FILE_H;
```

Now, some people aren't swayed by the sparkly nature of XML. They think that there's too much markup for each piece of content and would prefer something with fewer angle brackets. For these people there is a lighter-weight format called YAML (which stands for YAML Ain't Markup Language). YAML tries to strike a balance between structure and concision, and so it looks a little cleaner to the average eye:

```
---
name: agatha
address: 192.168.0.4
---
name: mr-tock
address:
    - 192.168.0.10
    - 192.168.0.11
    - 192.168.0.12
---
```

The Perl module to parse YAML[6] is called, strangely enough, YAML and is used like this:

6. One nice property of YAML is it is language-independent. There are YAML parsers and emitters for Ruby, Python, PHP, Java, OCaml, and even Javascript.

```
use YAML;
my @config = YAML::LoadFile('config.yml');
# @config now contains a list of references to hashes, one per record
# we now can use $config[N]->{address}
# (later...) dump the config back out to a file
YAML::DumpFile( 'config.yml' , @config );
```

If you'd prefer a more object-oriented way of working with YAML, Config::YAML can provide it.

There are an infinite number of possible formats for config files, but at least now we've hit the highlights.

All-in-One Modules

If all of this talk about picking the right module for config parsing has made your brain hurt, let me ease us toward the end of this article with a quick look at a set of modules which can help sidestep the choice.

Config::Context is a wrapper around the Config::General, XML::Simple, and Config::Scoped modules that allows you to use a single module for each of the formats those modules handle. On top of this, it also adds contexts à la Apache so you can use <Location> </Location> tags in those file formats.

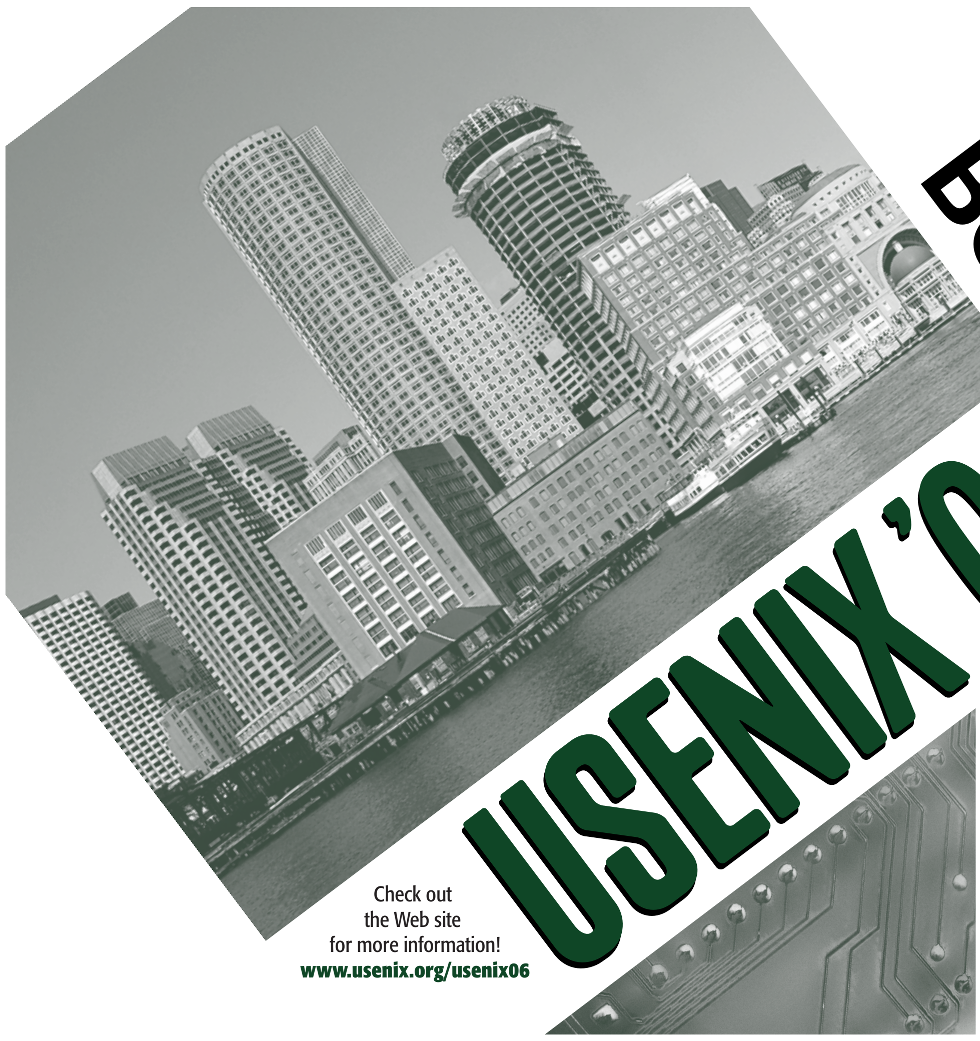
If you crave a module with a larger menu of config file formats supported, Config::Auto can handle colon/space/equals-separated key/value pairs, XML formats, Perl code, .ini formats, BIND9 style, and irssi config file formats. Not only that, it will (by default) guess the format it is parsing for you without further specification. If that's too magical for you, a format can be specified.

Epilogue

If you are sick of talking about config files at this point (I don't blame you), let's end with a brief mention of some of the more advanced alternatives. There are a number of other reasonable places to stash config information.[7] Shared memory segments can work well when performance is the key criterion. Many systems are now keeping their configuration in databases. Others have a specific network server to distribute configuration information.

These are all interesting directions to explore, but I'm afraid we're out of time. Take care, and I'll see you next column.

7. There are also a number of other unreasonable places—for example, hidden in image files using Acme::Steganography::Image::Png or in a play via Acme::Playwright.



Annual Technical Conference

Join us in Boston for 5 days of groundbreaking research and cutting-edge practices in a wide variety of technologies and environments.

Don't miss out on:

- **Extensive Training Program featuring expert-led tutorials**
- **New! Systems Practice & Experience Track (formerly the General Session Refereed Papers Track)**
- **Invited Talks by industry leaders**
- **And more**

Please note: USENIX '06 runs Tuesday–Saturday.

Check out the Web site for more information!
www.usenix.org/usenix06