DAVID N. BLANK-EDELMAN

practical Perl tools: a little place for your stuff

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Perl for System Administration*. He has spent the past 22+ 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 program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

> Actually this is just a place for my stuff, ya know? That's all, a little place for my stuff. That's all I want, that's all you need in life, is a little place for your stuff, ya know? I can see it on your table, everybody's got a little place for their stuff. This is my stuff, that's your stuff, that'll be his stuff over there. That's all you need in life, a little place for your stuff.
>
> —*George Carlin*

GIVEN THAT THIS IS THE FILESYSTEM and storage issue, let's take a look both at some of the most widely used ways of keeping your stuff using Perl and some of the less conventional methods you may have missed. We'll start with the most specific kind of stuff and move toward the most general.

Storing Your Perl Data Structures

There are times when your program has worked hard to create such a really useful data structure in memory that you want it to persist beyond the life of that current program run. Maybe your program runs once a week and wants to add that week's data to the previous runs. Maybe you have a job that is meant to run a long time before producing an answer, and (as they would say on *The Sopranos* if it took place in a datacenter), "You wouldn't want anything to happen to that data, say, right in the middle of the run, now would you, palsie?" You owe it to yourself to periodically dump important data structures and other context to disk so it is possible to resume the computation should power, net, or the little hamster that runs around in the machine doing the actual work cut out.

The most conventional approach to this problem is to use the Storable module that ships with Perl. It's really easy to use. To write a data structure to a file you can use:

```
use Storable;
my %datastructure = ...
# some convoluted data structure
nstore (\%datastructure, 'persistfile')
    or die "Can't write to persistfile\n";
```

Then, in another program (or another routine of the current program), you can use:

```
$dsref = retrieve('persistfile');
```

and you'll get back a reference to the data structure you stored. If you wanted to work with just a hash like the original data structure we stored instead of

a reference to it (and you don't mind the hit to copy the data), you can dereference it as per usual like this:

```
%datastructure = %{$dsref};
```

One last note about this code. I tend to use `nstore()`, as shown here, versus the default `store()` method, because it keeps the data in "network order" format. Storable writes its data in a binary format. Using the network order storage function ensures that this data can be read on different machines with different byte orders. It is a little less efficient to store in this format, but retrieving is still as fast and you gain more portability for your data.

That last note offers a good segue to a less common approach. If you are concerned about keeping your data around in some binary format because you are a Long Now kind of person (www.longnow.org) and you don't trust you'll have anything with which to read it back when Perl is at version 8.5, then a text-based data serialization module may be more your cup of tea. You could use something like Data::Dumper (or, better yet, the more obscure Data::Streamer) to write text to a file, but I'd assert you will be better off finding the most standard standard you can find and writing that format. Two such similar standards that fit the bill nicely are YAML (www.yaml.org) and JSON (www.json.org). The former has deeper Perl roots; the latter is well known because of its ties to Javascript and the AJAX world. The YAML folks also note that JSON is essentially a subset of YAML and so any decent YAML parser worth its salt should also be able to parse JSON.

In previous columns I've demonstrated the use of YAML, so let's take a quick look at JSON.

Taking a data structure and turning it into JSON is easy:

```
use JSON;
my %datastructure = ... # some convoluted data structure

  my $json = JSON->new();
  my $encoded = $json->encode(\%datastructure);
  # or, for prettier but less space-efficient results:
  # my $encoded = $json->pretty->encode(\%datastructure);
```

If we had a data structure like this:

```
%datastructure = ( 'Fred' => 1, 'Barney' => 2, 'Wilma' => 3 )
```

then $encoded would contain:

```
'{

  "Wilma" : 3,
  "Barney" : 2,
  "Fred" : 1
}'
```

which isn't all that impressive until you start playing with more complex data structures. I recently had a case where I needed to parse the output of a Java program that spat out JSON and I was able to write code like this:

```
use JSON;

# get the results of the zmGetUserFolders command in JSON format
open my $ZMMBOX, zmGetUserFolders($from_user) . "|"
  or die "Can't run zmGetUserFolders(): $!";
my $json_data = join( '', <$ZMMBOX> );
close $ZMMBOX;

# parse it into a Perl data structure
my $folders = from_json($json_data);
```

```
# extract the hrefs that contain the folders we care about
my @sharedfolders
  = grep { exists $_->{ownerId} and $_->{view} eq 'appointment' }
  @{ $folders->{children} };
```

This code ran the command, parsed the JSON output, and then walked the list of folders returned, looking for those with the right owner and view type.

Storing Key–Value Pairs

A slightly more general and hence more widely used scheme for data storage involves a simpler key–value model. For instance, Username is associated with "dnb," "FirstName" with "David," and "LastName" with "Blank-Edelman." It isn't a particularly sophisticated idea, but it is the thing that makes Perl's hashes (and Snobol4's tables, which led to associative arrays in awk to give the predecessor languages their proper due respect) so useful.

The easiest and most conventional way to store data like this is to use Perl's `tie()` functionality to call an external database library such as gdbm or BerkeleyDB (my preference). Two past columns discussed `tie()` in all of its glory, so let's make do with a really small example sans explication:

```
use BerkeleyDB; # this module has lots of firepower, see the doc for details
tie my %tiedhash, 'BerkeleyDB::Hash', -Filename => 'data.db'
  or die "Can't open data.db: $!\n";

$tiedhash{uid}       =  'dnb';
$tiedhash{FirstName} =  'David';
$tiedhash{LastName}  =  'Blank-Edelman';

untie %tiedhash;
```

Next time you tie to that database, you can retrieve the values you want for that key using standard Perl hash syntax.

Let's see two less conventional ways to deal with key–value pair storage. The first is to use a very cool module you may not have seen before. DBM::Deep is an almost entirely pure Perl module that implements an entire database engine. This engine stores its data in a portable format, actually implements ACID transactions, and is pretty darn fast even with very large datasets.

(Note that the module used to be entirely written in Perl until a recent revision started to depend on the FileHandle::Fmode module. If you look at the CPAN bug reports you'll find a pure-Perl replacement for that dependency if this is important to you.)

DBM::Deep can be used via `tie()` just like in our BerkeleyDB example, but then you'll lose an additional piece of magic: multi-level array and hash support. Yup, we're essentially combining the previous sections of this article with the current one because we can now write:

```
use DBM::Deep;

my $dbdeep = DBM::Deep->new('data.db');

$dbdeep->{uid}     =  'dnb';
$dbdeep->{name}  =  {  'FirstName'  =>  'David',
                       'LastName'  =>  'Blank-Edelman'};
```

Then in another program (or another part of the same program during a different program run), you can write:

```
print $dbdeep->{name}->{LastName}
```

and it will retrieve and print my last name. Pretty nifty!

The second, less conventional approach I want to mention but not really delve into is less of a storage technique and more of an optimization technique. If you have data that you only need to keep for a short amount of time (e.g., Web sessions) or have accessible in memory just while it is actively used, you should take a look at the various caching frameworks available. Cache::Cache is the most heavily used, but there are others. The CHI framework in particular seems to be up and coming and worth considering. Many of these frameworks will automatically serialize more complex data structures for you when you attempt to store that data similar to what Storable will do for you.

With all of these frameworks, you basically set up the kind of cache you want to use (in memory, stored as files, in shared memory, using a separate custom daemon such as memcached, etc.) and what flavor of cache you want (e.g., should the cache keep itself under a certain size?). Once you've picked your cache "backend" you then have the opportunity to add things to the cache using some sort of `set()` operator. This set operator usually lets you specify the amount of time that data should persist. Expired data will be flushed from the cache either automatically or at your command. To use the cache itself, you tend to write code that looks like this:

```
is item in cache?
   yes: retrieve and use data
   no: work harder to get data (pull from database, make a new one, etc),
      store the data in the cache and then return the new value
do stuff
purge the cache when the main work is done
```

Modules such as those in Cache::Cache and CHI handle all of the behind-the-scenes work of maintaining the cache for you, and everyone wins.

Storing Stuff in SQL Databases

If you mention SQL and databases in the same sentence, Perl programmers will reflexively just say "DBI." The "DataBase Independent interface for Perl" (dbi.perl.org) is one of the great gifts Perl, via Tim Bunce, has given the world.

It is basically an API that allows you to write database-agnostic code that will work independently of whatever database engine you are using today. It is a tremendous relief to be able to write code that will work unchanged with Oracle, MS-SQL, MySQL, Postgres, etc. Simple DBI code looks like this:

```
use DBI;

my $uid = $ARGV[0];

my $dbh = DBI->connect(
   "DBI:mysql:database=usenix;host=localhost", 'user', 'password')
      or die "Couldn't connect to database: " . DBI->errstr;

my $sth = $dbh->prepare('SELECT * FROM users WHERE uid = ?')
   or die "Couldn't prepare statement: " . $dbh->errstr;
$sth->execute($uid)
   or die "Couldn't execute statement: " . $sth->errstr;

my @results = ();
while (@results = $sth->fetchrow_array()){
   print join ('\n',@results);
}
```

```
$sth->finish;
$dbh->disconnect;
```

We connect to a specific database by providing the database engine name, database name, server host name, and authentication information. This returns a database handle through which we'll communicate with that database. We then pre-parse the SQL statement we're going to send using `prepare()`.(Although this last step is not strictly necessary, it will prove useful for better performance when our code gets more sophisticated.) This yields a statement handle. The statement handle will give us a means to run that query with `execute()` and return the result via `fetchrow_array()`. `fetchrow_array()` returns the results of that query to an array, one result row at a time. Once done, we close both our statement and database handles and we're done.

There's much more we could look at around DBI (several published books' worth, to be exact) but we're going to leave that behind so we can look at a lesser-known twist on this approach: DBD::SQLite.

All DBI-compatible database engines have a database-dependent driver called a DBD written to use them. The particular DBD I'd like to call your attention to is called DBD::SQLite. It not only provides the driver for the SQLite database engine (www.sqlite.org) but it also builds the actual runtime libraries it needs so you don't have to install anything extra to start using it. SQLite is a really wonderful lightweight SQL database engine that does not require a server to run. In some ways it is like the BerkeleyDB libs mentioned earlier, except that one can actually throw a fairly decent subset of SQL at it and it will do the right thing.

What does this mean to you? You can write reasonably portable Perl code using DBI and SQL without a server. Your database will live in a single file on your machine. This is wonderful for prototyping code or creating small projects that don't need the power a full database server would bring (and the concomitant hassles of setting it up).

Code that uses DBD::SQLite looks like any other DBI code. Instead of:

```
my $dbh = DBI->connect(
  "DBI:mysql:database=usenix;host=localhost", 'user', 'password')
    or die "Couldn't connect to database: " . DBI->errstr;
```

you write:

```
my $dbh = DBI->connect("dbi:SQLite:dbname=datafile.sql3", '', '',)
  or die "Couldn't connect to database: " . DBI->errstr;
```

and everything proceeds as normal from there.

And with that tip, I'm running out of, err, space. Take care, and I'll see you next time.