

The Perl Journal

Internet Socket Programming Using Perl

Thomas Valentine • 3

Practical Secure Port Knocking

John Graham-Cumming • 5

Building GUIs with *Win32::GUI::XMLBuilder*

Blair Sutton • 8

Using the Web as a GUI

Simon Cozens • 12

Scripts as Modules

brian d foy • 16

PLUS

Letter from the Editor • 1

Perl News by Shannon Cochran • 2

Book Review by Jack J. Woehr:

***The Perl CD Bookshelf Version 4.0* • 19**

Source Code Appendix • 21

LETTER FROM THE EDITOR

The Time-Tested Widget

When measuring an application's usefulness, it's hard to overstate the importance of the interface. Sure, your subroutines have to do their job, and your code needs to run reasonably fast, and certainly, it needs to not crash. But if users are going to get any real work done, an application's interface has to make sense.

Case in point: A certain app that I use almost daily (which shall remain nameless) violates just about every interface principle I've ever heard of, much to the program's detriment. Under the hood, this app has world-class processing, but good luck trying to find out how to make use of it. I could catalog all its faults, but there's one interface bungle that sort of sums up the entire app's GUI ineptitude—it has several interface widgets that perform completely different functions when clicked on their left and right halves, though they have no visible markings distinguishing the left half from the right half of the widget. Gems like this make me wonder if the designers aren't having a bit of a laugh at users' expense.

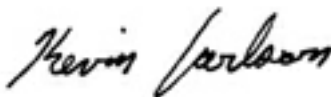
But enough criticizing. There's a good reason why these sorts of interfaces persist in the software world: *Interface design is hard*. Interface designers break the rules for what seem like good reasons. They create new widgets because they find that, in many toolkits, supplied widgets don't do enough. All too often, though, these new widgets make little sense to users because they merely visually encode some idiosyncratic notion that exists mainly in the designer's head. Here's a tip: If your new widget requires three paragraphs in the manual to explain its use, you've blown it.

GUI toolkits exist in the first place because some group has communally decided that certain interface "best practices" are universal enough that we should make them standard. You can argue the relative merits of some of the widgets and concepts present in various GUI toolkits, but some things are pretty universal—we all understand the function of buttons and checkboxes, for instance. Of course, there are geniuses out there who can carry off a revolutionary interface design, but I, not being confident in my status as genius, will stick to the established concepts.

One good way to confine yourself to a safe GUI sandbox is to make your app web-based. HTML has a wonderful way of limiting you to well-understood interface conceits. Yes, it's limiting in other ways, too, but often those limitations are a price worth paying, as they can force you to simplify your design. In his column this month, Simon Cozens makes good use of HTML, and packages up a browser-based interface, a simple web server, a web app, and a browser into a single executable using PAR, Perl's answer to Java's JAR. It's a great technique that eliminates many dependencies that can make it very hard to ship cross-platform Perl apps.

Also in this issue, Blair Sutton makes use of standard *Win32::GUI* interface elements, further abstracting them into XML files that can be more easily manipulated than standard *Win32::GUI* objects.

Neither Simon nor Blair buck any interface trends with these techniques. They don't reinvent any wheels. They simply do what Perl folks everywhere do everyday—they stand on the shoulders of giants to write code we can all understand and make use of.



Kevin Carlson
Executive Editor
kcarlson@tpj.com

Letters to the editor, article proposals and submissions, and inquiries can be sent to editors@tpj.com, faxed to (650) 513-4618, or mailed to *The Perl Journal*, 2800 Campus Drive, San Mateo, CA 94403.

THE PERL JOURNAL is published monthly by CMP Media LLC, 600 Harrison Street, San Francisco CA, 94017; (415) 905-2200. SUBSCRIPTION: \$18.00 for one year. Payment may be made via Mastercard or Visa; see <http://www.tpj.com/>. Entire contents (c) 2004 by CMP Media LLC, unless otherwise noted. All rights reserved.



The Perl Journal

EXECUTIVE EDITOR

Kevin Carlson

MANAGING EDITOR

Della Wyser

ART DIRECTOR

Margaret A. Anderson

NEWS EDITOR

Shannon Cochran

EDITORIAL DIRECTOR

Jonathan Erickson

COLUMNISTS

Simon Cozens, brian d foy, Moshe Bar, Andy Lester

CONTRIBUTING EDITORS

Simon Cozens, Dan Sugalski, Eugene Kim, Chris Dent, Matthew Lanier, Kevin O'Malley, Chris Nandor

INTERNET OPERATIONS

DIRECTOR

Michael Calderon

SENIOR WEB DEVELOPER

Steve Goyette

WEBMASTERS

Sean Coady, Joe Lucca

MARKETING / ADVERTISING

PUBLISHER

Michael Goodman

MARKETING DIRECTOR

Jessica Hamilton

GRAPHIC DESIGNER

Carey Perez

THE PERL JOURNAL

2800 Campus Drive, San Mateo, CA 94403
650-513-4300. <http://www.tpj.com/>

CMP MEDIA LLC

PRESIDENT AND CEO Gary Marshall

EXECUTIVE VICE PRESIDENT AND CFO John Day

EXECUTIVE VICE PRESIDENT AND COO Steve Weitzner

EXECUTIVE VICE PRESIDENT, CORPORATE SALES AND MARKETING

Jeff Patterson

CHIEF INFORMATION OFFICER Mike Mikos

SENIOR VICE PRESIDENT, OPERATIONS Bill Amstutz

SENIOR VICE PRESIDENT, HUMAN RESOURCES Leah Landro

VICE PRESIDENT/GROUP DIRECTOR INTERNET BUSINESS

Mike Azara

VICE PRESIDENT AND GENERAL COUNSEL Sandra Grayson

VICE PRESIDENT, COMMUNICATIONS Alexandra Raine

PRESIDENT, CHANNEL GROUP Robert Faletra

PRESIDENT, CMP HEALTHCARE MEDIA Vicki Masseria

VICE PRESIDENT, GROUP PUBLISHER APPLIED TECHNOLOGIES

Philip Chapnick

VICE PRESIDENT, GROUP PUBLISHER INFORMATIONWEEK

MEDIA NETWORK Michael Friedenberg

VICE PRESIDENT, GROUP PUBLISHER ELECTRONICS

Paul Miller

VICE PRESIDENT, GROUP PUBLISHER NETWORK COMPUTING

ENTERPRISE ARCHITECTURE GROUP Fritz Nelson

VICE PRESIDENT, GROUP PUBLISHER SOFTWARE DEVELOPMENT

MEDIA Peter Westerman

VP/DIRECTOR OF CMP INTEGRATED MARKETING SOLUTIONS

Joseph Braue

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Shannon Aronson

CORPORATE DIRECTOR, AUDIENCE DEVELOPMENT

Michael Zane

CORPORATE DIRECTOR, PUBLISHING SERVICES Marie Myers

Perl News

Perl 6: Not Here Yet...

Larry is archiving the most recent versions of the Apocalypses at <http://www.wall.org/~larry/apo/>, and the Synopses at <http://www.wall.org/~larry/syn/>. As Larry noted on the perl6-language list, recent changes include:

- Strictures and warnings are now the default in the main program.
- Syntax of subtype and enum declarations are more like normal declarations.
- Subtypes are now declared with the *subtype* keyword.
- The symbolic ref syntax reverted back to `::($expr)`.
- The syntax of operators are now defined to leverage hash subscript syntax:

```
infix:<<+>>      # op name always quoted now
circumfix:<<[ ]>> # multipart ops are now
                  # slices, no ... needed
circumfix:{'[','']}> # same thing
```

- Special rule for splitting symmetrical operator names is gone; use slice.
- Special rules for recognizing the ends of operator names are gone.
- Current class is `?$CLASS` or `::?CLASS`, not `::_`.
- Current sub is `?$SUB`, not `&_`.
- There is no bogus first argument to the pair-arg variant of `bless`.

...But the Perl Dev Kit 6 Is

ActiveState has updated its Perl Dev Kit to version 6.0, introducing graphical user interfaces for most tools and visual guides to build options. Two new bundles are also included—PDK Deployment Tools and PDK Productivity Tools. According to ActiveState, the PDK Deployment Tools offer “new graphical user interfaces for all application builders to simplify the creation of Perl executables. Technical enhancements include shared library options and a dynamic DLL loader. The deployment tools also feature PerlMSI, for creating Microsoft MSI installation files.” The PDK Productivity Tools include a graphical debugger, Visual Package Manager “for administering Perl installations locally or remotely,” and the new VBScript Converter: “Capable of recognizing all popular VBScript type libraries, including Microsoft Excel, Windows Scripting Host, Lotus Notes and more, the converter allows developers to leverage the huge VBScript libraries available on MSDN, other web sites, or in-house, by converting them to Perl code for integration into larger applications. Alternatively, users can record VBA macros with the Microsoft Office macro recorder, then translate them to Perl.”

Also new is the PDK Filter Builder, “for interactively creating text processors that do custom filtering and replacement operations.

It can be used as a pure visual tool by simply outputting modified text, or to generate Perl code that performs the filtering, for adding to a toolkit or embedding into applications. Filter Builder displays the filtered results on the fly, allowing rapid iterations and...what-if interactions.” See <http://www.activestate.com/PDK> for details.

For the Birds

Matt Fowles has succeeded Piers Cawley as summarizer of the Perl 6 mailing lists, heroically undertaking to recap all three lists. In his inaugural report (<http://www.perl.com/pub/a/2004/10/p6pdigest/20041017.html>), he pointed to a thread advising newcomers seeking to become involved in Perl 6 development: “Pratik Roy wondered if he could join into the work on Parrot or if he would remain an outsider looking in. Many people rushed to suggest ways in which he could help. So, please don’t feel afraid to contribute. At worst, a patch of yours will be turned away with an explanation as to why. At best, you will start being referred to by first name only in the summaries.” The advice for Pratik included: writing tests; writing documentation; checking the TODO list at <http://www.parrotcode.org/todo.html>; and working on the implementation of Python on Parrot.

A related effort, Michael Pelletier’s “Parrot Forth” language—which he calls Parakeet—has advanced to version 0.3 and been added to CVS. As Michael explains: “Parakeet is a stack language like Forth. You could call it Forth by many people’s definition, but it *isn’t* standard Forth. Parakeet is Forth that is extremely machine specific to the Parrot VM. Parakeet also has a lot of features not found in standard Forth, like local variables, nested words and classes and (as a result) nested compile-time and runtime lexical word, class and variable scopes. Parakeet is fully dynamic and unprototyped; all variables and bindings are looked up at runtime using Parrot’s lexical scratchpad stacks. In many ways its level of object dynamics approaches that of Python, while still retaining that Forth minimalistic thing. Parakeet is written in PIR and compiles new words directly to PIR. There is no ‘inner loop’ above the Parrot VM and core words are not ‘threaded’ (in the Forth sense of the word), they are directly inlined into the new word’s body.”

Toronto Perl Mongers Post Audio Archives

The Toronto Perl Mongers group has posted audio recordings of its sessions at http://hew.ca/talks_audio/. You can find sessions there on generating passwords; using Zope and Plone; the `WWW::Mechanize`, `HTML::TokeParse`, and `Getopt::Declare` modules; and automated testing with `Class::DBI`, among many other topics. In his announcement to perlmonks.org, Fulko Hew added, “I also hope to encourage the Perl community to create a central repository where we can all archive this stuff. A place with a lot of disk space, and a lot of bandwidth. If anyone has suggestions, or can volunteer such a web server, let me know.”

Internet Socket Programming Using Perl

The role of Perl on the Internet is hard to ignore. Some would make the case that Perl is the most used and most capable scripting language on the Internet today, and this is probably true. Its versatility and ease of use has made it an industry standard, used throughout the world.

Perl sockets work like file handles that can span a network or the entire Internet. With sockets, you can communicate with any computer that is connected via the Internet or via a network by specifying the server to connect to and a port to connect through. After the initial connection, you may use that connection just as you would a file handle. There are a few differences, but these are mainly academic.

To put it simply, the two sides of a Perl socket are the server and the client. It doesn't really matter which is which—the communication process is essentially the same because the client can send information to the server, and vice versa. The communication process is surprisingly simple. The process starts with the creation of the server. The server waits for a connection request from the client, which can be created any time after the creation of the server. The client connects to the server using the address of the server and the mutually agreed upon port number. If the port numbers are different, no connection is made. Once the client is connected, data may be sent from the server to the client, and vice versa.

There are two ways that data may be sent over a socket: two-way using one socket, or one-way using two sockets. In the one-socket method, data is sent from the server to the client and from the client to the server using one socket for both. In the two-socket method, one socket is used to send data from the server to the client and one is used to send data from the client to the server. I'll cover both methods in this article.

Sockets exist on all major operating systems. They are the accepted way to achieve communication between computers. Keeping this in mind, you can connect a UNIX computer to a Windows computer (and vice versa) with little or no trouble—if Perl is successfully installed on both machines. It should be noted that the ports used for an operation like this differ from OS to OS. Just keep in mind a few simple facts: Ports below 1024 are normally reserved for system use, so save yourself the headache and don't use them. HTTP requests usually use port 80, for example. Using a port number roughly in the 1025 to 5000 range usually produces

adequate results. Just be sure to use the same port on both computers being connected. Now, let's get to the practical coding.

Before you can connect to a server, or even create the server, you must know whether you can establish a connection on that port. To do this, use the *can_read* or *can_write* methods of the *IO::Select* module, which is included with Perl. The following example checks four sockets to see which ones are available for the connection.

```
Use IO::Select

$select = IO::Select->new();

$select->add($socket1);
$select->add($socket2);
$select->add($socket3);
$select->add($socket4);

@ok_to_read = $select->can_read($timeout);

foreach $socket (@ok_to_read) {
    $socket->recv($data_buffer, $flags)
    print $data_buffer;
}
```

This example uses the *can_read* method to check if any of the sockets may be used. You can just as easily use the *can_write* method in this particular example—the end result is the same. The script starts with the declaration of the use of the *IO::Select* module. A new instance named *\$select* is created, then the four sockets are created using the *add()* method. The sockets are placed into the *@ok_to_read* array, where the *can_read* method is invoked. A *foreach* loop is then used to print the results from the *can_read* call to the contents of the *@ok_to_read* array. The *recv* method is used to send info through the socket. Using the *recv* or *send* functions, you can send byte streams through your socket. You can also use the simple print and angle operators (< and >) to send text through the socket. Just be sure to use the newline character (*\n*) at the end of each line if you're sending text; the newline character is required to send the text string through the socket—if there is no newline character, no data is sent.

Now that you know how to check the sockets, we can delve into the creation of the common servers and clients that Perl is able to

Thomas is a writer residing in Selkirk, Manitoba, Canada. He can be contacted at publications@forkedweb.com.

use. We'll start with the creation of a TCP server and a TCP client using `IO::Socket`. Since both the server and the client work hand in hand, we'll cover them both at the same time. There are, however, a few bases that we have to cover—namely, the eight parameters that are passed to the `new()` method of `IO::Socket::INET`.

PeerAddr—The DNS address of the machine you'd like to connect to. The address may be the dotted IP address of the machine, or the domain name itself; "walkthegeek.com," for example.

Perl sockets work like file handles that can span a network or the entire Internet

PeerPort—The port on the host machine you'd like to connect to. We'll use port 2000, for no particular reason.

Proto—The protocol that will be used, the options being `tcp` or `udp`.

Type—The type of connection you'd like to establish, the options being `SOCK_STREAM` for tcp data, `SOCK_DGRAM` for udp connections, or `SOCK_SEQPACKET` for sequential data packet connections. We'll be using tcp, so the `SOCK_STREAM` connection type will be used.

LocalAddr—The local address to be bound, if there is one. We won't use one.

LocalPort—The local port number to be used.

Listen—The queue size, which is the maximum number of connections allowed.

Timeout—The timeout value for connections.

All socket communications using `IO::Socket` use these same parameters. We'll use the `new()` method of `IO::Socket::INET` to return a scalar that holds a filehandle, which is an indirect file handle type. Let's make a client:

```
use IO::Socket;

$socket = IO::Socket::INET->new
(
    PeerAddr => 'walkthegeek.com',
    PeerPort => '2000',
    Proto => 'tcp',
    Type => 'SOCK_STREAM'
) or die "Could not Open Port.\n";
```

So there you go. You've made a TCP client using the sockets of `IO::Socket::INET`. Now let's write to the server from the client. The connection coding is the same, so it doesn't need to be repeated. As in the previous example, the port used is 2000, the server is walkthegeek.com, the protocol is tcp, and the type is `SOCK_STREAM`:

```
print $socket "Hello there!!\n";

close ($socket);
```

The example sends a text message to the server. The message "Hello there!!" will be displayed on the server's console. Notice that the newline character (`\n`) has been dutifully used on the end of the print statement.

To read from the server, the creation of the socket and the subsequent connection is the same, and only the print statement changes:

```
$answer = <$socket>;

print $answer;

close ($socket);
```

The above code pulls the message from the server, displaying it on the console of the client. Creating a server is as easy as creating the client. The same connection coding is used, only the `$socket` is replaced with `$server`:

```
$server = IO::Socket::INET->new
(
    PeerAddr => 'walkthegeek.com',
    PeerPort => '2000',
    Proto => 'tcp',
    Type => 'SOCK_STREAM'
) or die "Could not Open Port.\n";
```

What is different in the creation of the client as opposed to the creation of the server is the coding below the connection snippet—a `while` loop is used. Here's an example showing the server reading from the client:

```
while ($client = $server->accept())
{
    $line = <$client>;
    print $line;
}

close ($server);
```

The `while` loop calls the `accept()` method, which makes the server wait for a client to connect. The body of the `while` loop is executed when the client connects, reading (pulling) any messages from the client. To write to the client, the only thing that changes is the body of the `while` loop:

```
while ($client = $server->accept())
{
    print "Hello Client !!\n";
}

close ($server);
```

In the examples so far, I've used `tcp` for the sockets. You can just as easily use `udp`, however. Just substitute `udp` for `tcp` in the `Proto` parameter, and use `SOCK_DGRAM` in place of `SOCK_STREAM` for the `Type` parameter.

In a nutshell, that's socket programming. With this knowledge, you can create chat programs, run a multiplayer game over the Internet, and just generally have fun. Some working, practical examples of the use of sockets are the Yahoo Messenger, MSN Messenger, and various other Java and Perl-based chat programs.

Practical Secure Port Knocking

Last year, Martin Krzywinski described a technique for stealthily communicating with a computer (see “Port Knocking: Network Authentication Across Closed Ports,” *Sys Admin* magazine, June 2003). The idea was that open ports on a machine invite attack. If you leave a machine on the Internet with an SSH daemon running on port 22, it’s a simple matter for attackers to use port scanners like nmap (<http://www.insecure.org/nmap/>) to discover that the port is open and then try to attack it.

Krzywinski suggested that sensitive ports should be left closed until opened using a secret knock. The knock consists of sending TCP SYN packets (the first packet sent when opening a TCP connection) to a sequence of closed ports on the target machine. Firewall software records in a log file the failed connection attempts and software watching the firewall log checks for a specific sequence of ports and enables/disables a service. For example, with the correct knock, an SSH daemon could start waiting for connections, or the firewall could be reconfigured to allow connections from the host that knocked.

Since a knock is just a sequence of attempted TCP connections, programs such as telnet can be used to manually generate it. For example, to knock on ports 42, 196, and 69 of 192.168.0.3, do:

```
telnet 192.168.0.3 42
telnet 192.168.0.3 196
telnet 192.168.0.3 69
```

Imagine that Linux host 192.168.0.3 is running an SSH daemon on port 22, but the iptables firewall (<http://www.netfilter.org/>) has been set to drop and log every incoming TCP packet, thus making access to the SSH daemon impossible:

```
iptables -N LOGNDROP
iptables -A INPUT -p tcp -j LOGNDROP
```

John is vice president of engineering at Electric Cloud, which focuses on reducing software build times. He can be contacted at jgc@electric-cloud.com.

```
iptables -A LOGNDROP -j LOG --log-level warn \
--log-prefix='FIREWALL:'
iptables -A LOGNDROP -j DROP
```

This tells iptables to create a new chain called *LOGNDROP*, then pass all incoming TCP packets to the chain where they are first logged (most likely to */var/log/messages*) with the prefix *FIREWALL:* (for easy grepping), then dropped.

The use of the action *DROP* means that TCP packets are not acknowledged in any way. The packet is simply discarded. The host machine will not even generate an ICMP “port unreachable” packet and the host appears to be switched off, but the connection attempt has been logged. To outsiders, the system appears to be inoperative, but the log file tells a different story.

An application watching */var/log/messages* sees entries such as those in Example 1 if a knock is made against ports 42, 196, and 69 from host 192.168.0.5. The same application could process the ports knocked to decide to add or remove entries in iptables, thereby opening or closing ports. For example, the correct knock on 42, 196, 69 could result in the firewall opening port 22 just for host 192.168.0.5 for SSH access with the iptables command:

```
iptables -I INPUT -p tcp -s 192.168.0.5 --
dport 22 -j ACCEPT
```

Port knocking has three fundamental ideas behind it:

- **Default to Closed.** It’s better to leave sensitive services firewalled until they are needed so that scanning the machine with nmap doesn’t reveal any ports to attack.
- **Share a Secret.** How ports get opened should rely on a secret (for example, a password in the form of a specific sequence of ports).
- **Vow of Silence.** The port knocking application does not respond to any packets. It listens but divulges no information about its operation, or even existence, so that it too is undetectable with nmap. A slightly more controversial tenet of port knocking is that the

knock should be hard to intercept. Some people accuse port knocking of relying on “security through obscurity”: that is, both the sequence of ports to knock is secret and it’s not possible to determine the sequence.

Obviously the simplistic port knocking system just outlined has vulnerabilities and it is possible to detect the knocks using a packet sniffer. Eavesdroppers can easily detect the knock sequence using, for example, `tcpdump` (<http://www.tcpdump.org/>) with an appropriate filter:

```
tcpdump -t -n '(tcp[13] == 2) or (tcp[13] == 18)'
```

which prints out all TCP SYN and SYN/ACK packets. For example, intercepted knocks on 42, 196, and 69 would look like Example 2. A Perl script (Listing 1) can quickly parse `tcpdump`’s output to determine which SYN packets received a SYN/ACK (hence, were real connections) and which were silently dropped (and hence could be a knock), then produce the output:

```
Knock on 192.168.0.3:42
Knock on 192.168.0.3:196
Knock on 192.168.0.3:69
```

Once the knock is known, it can be repeated by a third party to open the SSH port for their use at any later time.

Because of this vulnerability, encryption of the knock is essential (for more information on encryption techniques, see <http://www.portknocking.org/>). The knock needs to be nonreplayable (that is, eavesdroppers can’t reuse the knock for their own use), nonspoofable (eavesdroppers can’t reuse the knock from a different IP address), and should not be easily decodable.

Tumbler

The Tumbler protocol (named for the parts of a lock that tumble into place when the right key is inserted) implements the spirit of port knocking with robust security using a well-known hashing algorithm. Tumbler provides protection against replay attacks (the knock is timestamped and cannot be reused after a short interval) and spoofing (the knock can only be used from a specific IP address).

In addition to the protocol, a Perl implementation is available under the General Public License at <http://tumbler.sf.net/>. Of course,

```
Aug 11 10:00:46 kernel: FIREWALL:SRC=192.168.0.5 DST=192.168.0.3
LEN=60 TOS=0x10 PREC=0x00 TTL=64 PROTO=TCP SPT=32769 DPT=42
Aug 11 10:00:55 kernel: FIREWALL:SRC=192.168.0.5 DST=192.168.0.3
LEN=60 TOS=0x10 PREC=0x00 TTL=64 PROTO=TCP SPT=32770 DPT=196
Aug 11 10:00:57 kernel: FIREWALL:SRC=192.168.0.5 DST=192.168.0.3
LEN=60 TOS=0x10 PREC=0x00 TTL=64 PROTO=TCP SPT=32771 DPT=69
```

Example 1: Log entries (emphasis added).

```
192.168.0.5.32772 > 192.168.0.3.42: S 3876367475:3876367475(0) win
5840 <mss 1460,sackOK,timestamp 10426325 0,nop,wscale 0> (DF)
192.168.0.5.32774 > 192.168.0.3.196: S 3902950029:3902950029(0) win
5840 <mss 1460,sackOK,timestamp 10434715 0,nop,wscale 0> (DF)
192.168.0.5.32773 > 192.168.0.3.69: S 3883917621:3883917621(0) win
5840 <mss 1460,sackOK,timestamp 10431805 0,nop,wscale 0> (DF)
```

Example 2: Intercepted knocks.

```
my ($sec,$min,$hour,$mday,$mon,$year) = gmtime(time);
my ($my_port,$my_ip) = sockaddr_in($socket->sockname);
my $me = inet_ntoa($my_ip);
my $hash = sha256_hex("$year$mon$mday$hour$min:$me:$secret");
$socket->send("TUMBLER1: $hash");
```

Example 3: A script named “tumbler” sends the knock.

there’s no reason why Tumbler has to be implemented in Perl—the protocol is simple and could easily be built into other applications (for example, SSH could include a `--tumbler` option to perform the appropriate knock before connecting).

The protocol consists of a single UDP packet in the form:

```
TUMBLER<v>: <knock>
```

where `<v>` is the protocol version number, currently 1, and `<knock>` is a hexadecimal string containing the knock. For example, the UDP packet might contain:

```
TUMBLER1:844c17eee03d848cc0a60e90f608d5ea11f417d
9bf0d2c1af2b5
```

There is no response to the message. Either the process listening for Tumbler messages accepts it or it is silently dropped.

The `<knock>` is created by hashing the following three pieces of information using the SHA256 algorithm (<http://csrc.nist.gov/CryptoToolkit/tkhash.html>):

- The current Zulu date/time in the format YYYYMMDDHHmm (YYY is the number of years since 1900, MM is the month starting with January as 0, DD is the day of the month starting with 1, HH is the Zulu hour in a 24-hour clock, mm is the Zulu minutes).
- The dotted decimal representation of the sender’s IP address (192.168.0.5, for example).
- A shared secret password string.

SHA256 is used because it is a known secure hash algorithm (in fact, it’s an NIST Standard) that produces a one-way hash of its plain text. Because Tumbler only needs to recognize the validity of a message, and doesn’t need to decode it, a secure hash algorithm is appropriate. When a Tumbler message is received, the host creates a hash based on:

- The current Zulu date/time on the machine.
- The IP address of the person sending the Tumbler messages.
- The shared secret password string.

It then compares the two hashes to see if the message is valid. If the hash matches, then the host can proceed to open the firewall. If any part of the message is different (for instance, the time is wrong, the IP address doesn’t match, or the secret password is incorrect), then a different hash is generated and the host discards the Tumbler message.

If the host has multiple possible knocks, then each is configured with a different shared secret and the host runs through all the possible hashes looking for a match.

Implementation

In the Perl implementation (available online at <http://www.tpj.com/>), a script named “tumbler” sends the knock using Example 3 (with `$secret` containing the user’s secret password). It grabs the Zulu time by calling `gmtime`, gets the machine’s IP address in dotted format by calling `inet_ntoa` (on an `IO::Socket` called `$socket`), then hashes the message using `sha256_hex` (which is part of the Perl module `Digest::SHA`; <http://search.cpan.org/~mshelor/DigestSHA5.02/SHA.pm>) and returns a hex string containing the SHA256 hash. Finally, it sends a single UDP packet containing the Tumbler message.

The receiving machine runs another Perl script named “tumblerd” (also available online), which is configured through the configuration file `tumblerd.conf` to listen on a UDP port for Tumbler messages and perform commands. In Listing 2, *tumblerd* has been configured to allow SSH connections once a knock with the password *open-pAsSwOrD* has been made, and there’s even a knock that closes the port again (password *close-pAsSwOrD*).

For example, to open the SSH port, the remote user runs:

```
tumbler --open tumbler://open-pAsSwOrD@host:8675/
```

or alternatively:

```
tumbler --open tumbler://host:8675/
```

and then types in the secret password *open-pAsSwOrD*. The tumblerd runs through all the configured secrets looking for a hash match and, when it finds one, it runs the appropriate command.

A simple shell script containing three commands could establish an SSH connection to the host after opening the port with a knock and close the port again when the SSH connection is complete:

```
#!/bin/bash
tumbler --open tumbler://open-pAsSwOrD@host:8675/
ssh host
tumbler --open tumbler://close-pAsSwOrD@host:8675/
```

Security Properties

Tumbler clearly shares with port knocking the idea that ports are closed by default, and the use of a shared secret to open ports. To maintain the “vow of silence,” *tumblerd* does not respond to packets sent, but it is possible to discover the existence of *tumblerd* using *nmap* UDP scanning if the firewall is not configured to drop unknown UDP packets.

To run a UDP scan a host with *nmap*, type:

```
nmap -sU host -p1-65535
```

This detects the *tumblerd* daemon running:

```
Starting nmap 3.30 ( http://www.insecure.org/nmap/ )
Interesting ports on host (192.168.0.3):
Port State Service
8675/udp open unknown
```

nmap UDP scanning works by sending a packet to each port scanned and looking for the ICMP “port unreachable” message sent when

the port is closed. If there’s no port unreachable reply, then the port is open. To prevent that from happening, configure *iptables* to drop all UDP packets except those destined for *tumblerd*:

```
iptables -A INPUT -p udp --dport 8675 -j ACCEPT
iptables -A INPUT -p udp -j DROP
```

Dropping the packets means that they are silently discarded. Tumbler guards against replay attacks, where eavesdroppers intercept Tumbler packets and try to reuse them in two ways:

- Embedded in the hash is the Zulu time the knock was sent with an accuracy of one minute. The *tumblerd* daemon only accepts valid hashes; hence, the packet times out automatically.
- *tumblerd* automatically discards duplicate hashes. Not only is the packet useless in under a minute, repeated use of it within the same minute has no effect.

To guard against spoofing, where eavesdroppers use an intercepted knock from a different IP address, Tumbler includes the IP address of the sender in the hash. For attackers to spoof the Tumbler protocol, they must intercept a packet and reuse it in under 60 seconds from the same IP address as the packet was originally sent.

Since the *tumblerd* passes the IP address of the sender to commands that it executes, the attacker would only be able to repeat exactly the same command as the original knock.

Tumbler is easier to sniff than a TCP SYN-based port knocking implementation because, once a single Tumbler packet has been intercepted, the destination port is known and can be tracked using *tcpdump*:

```
tcpdump 'dst port 8675'
```

Other Implementations

Tumbler is just one implementation of the port knocking idea. In addition to reading <http://www.portknocking.org/>, check out the following interesting projects: *doorman* (<http://doorman.sourceforge.net/>) and *knockd* (<http://www.zeroflux.org/knock/>). Most interesting of all is *cd00r* (<http://www.phenoelit.de/stuff/cd00rdescr.html>), which was intended for use in malware. An academic look at port knocking comes in the form of a research paper from Intel Research (http://www.intelresearch.net/Publications/Berkeley/012720031106_111.pdf).

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
#!/usr/bin/perl -w
use strict;
my @knocks;
while ( <> ) {
    my $packet = $_;
    $packet =~ /^(?!\d+\.){3}\d+\.\d+ > (?!\d+\.){3}\d+\.\d+\/;
    my ($src_ip,$src_port,$dest_ip,$dest_port) = ($1,$3,$4,$6);
    if ( $packet =~ / ack / ) {
        @knocks = grep( !/^$src_ip:$src_port$/, @knocks );
    } else {
        push @knocks, "$dest_ip:$dest_port";
    }
}
foreach my $k (@knocks) {
    print "Knock on $k\n";
}
```

Listing 2

```
# The common section contains configuration options for the tumblerd daemon,
# here we set the UDP port to listen on to 8675 and a log file
```

```
[common]
port = 8675
log = /var/log/tumblerd.log
# Each door that a user can knock on is defined by a unique [door-X]
section,
# the first section is for opening the SSH port, and second for closing
#
# Each door has a secret (i.e. the password for this
# door that is part of the knock) and a command to execute.
#
# In the command it's possible to use the macros %IP% for the IP address of
# the person who knocked and %NAME% for the name of the door (in the
# first door here the name is open-ssh)
[door-open-ssh]
secret = open-pAsSwOrD
command = iptables -I INPUT -p tcp -s %IP% --dport 22 -j ACCEPT
[door-close-ssh]
secret = close-pAsSwOrD
command = iptables -D INPUT -p tcp -s %IP% --dport 22 -j ACCEPT
```


Building GUIs with *Win32::GUI::XMLBuilder*

I wrote *Win32::GUI::XMLBuilder* as a way to build simple Win32 GUI interfaces in Perl while keeping the visual design separate from the inner workings of the code. *Win32::GUI::XMLBuilder* makes it easy to add a Win32 user interface to whatever script or module you're writing. In this article, I will try to describe how to build a simple *Win32::GUI::XMLBuilder* application and also provide a simple template for doing this.

What Is It?

Win32::GUI::XMLBuilder (WGX) parses well-formed valid XML containing elements and attributes that help to describe and construct a *Win32::GUI* object. If you're unfamiliar with XML, I suggest you read the great introductory tutorial at <http://www.w3schools.com/xml/default.asp>.

Here is one way to use WGX:

```
use Win32::GUI::XMLBuilder;
my $gui = Win32::GUI::XMLBuilder->new(*DATA);
Win32::GUI::Dialog;
__END__
<GUI>
  <Window height='60'>
    <Button text='Push me!' onClick='sub
      {$_[0]->Text("Thanks")}' />
  </Window>
</GUI>
```

Or equivalently using two files:

```
# button.pl
#
use Win32::GUI::XMLBuilder;
my $gui = Win32::GUI::XMLBuilder->new({file=>
  "button.xml"});
Win32::GUI::Dialog;

# button.xml
#
<GUI>
  <Window height='60'>
    <Button text='Push me!' onClick='sub
      {$_[0]->Text("Thanks")}' />
```

Blair is head of software development for Odey Asset Management, a London-based hedge fund. He can be reached at bsdz@numenine.com.

```
</Window>
</GUI>
```

This will produce something like Figure 1, which changes to the dialog in Figure 2 when pressed.

Installing a Windows File Type Handler

If you plan to use WGX applications regularly, you could add a special file type to Windows Explorer—I personally like to use the extension WGX and configure the Run action to use the following application:

```
"C:\Perl\bin\wperl.exe" -MWin32::GUI::XMLBuilder -
e"Win32::GUI::XMLBuilder->new({file=>shift
@ARGV});Win32::GUI::Dialog" "%1" %*
```

This will allow WGX files to be automatically executed by double clicking on them or even by typing them in a Cmd.exe shell.

Mapping *Win32::GUI* to *Win32::GUI::XMLBuilder*

If you are familiar with *Win32::GUI*, you probably noticed that *Win32::GUI::Dialog* is the function that begins the dialog phase for a *Win32::GUI* object, which is what makes the GUI visible to the user. WGX includes *Win32::GUI* and all its subclasses, so any function in *Win32::GUI* is accessible from the code sections of a WGX mark-up file/program.

In fact, WGX might be considered just another way to write *Win32::GUI* code, where *Win32::GUI* objects become XML elements and *Win32::GUI* options become XML attributes. See Table 1.

Elements that will require referencing in your code should be given a name attribute, such as:

```
<Button name='MyButton' />
```

You can then access this widget internally within your XML as *\$self->{MyButton}*. If you constructed your WGX using something like this:

```
my $gui = Win32::GUI::XMLBuilder->new(...);
```

then you can access it as *\$gui->{MyButton}* in your script. If you do not explicitly specify a name, one will be created. Attributes can contain Perl code or variables, and generally any attribute that contains the variable *\$self* or starts with *exec:* will



Figure 1: A simple dialog built with Win32::GUI::XMLBuilder.



Figure 2: The dialog from Figure 1 after the button is pressed.



Figure 3: A window created with `<Window width='100' height='50'/>`.

	Win32::GUI	Win32::GUI::XMLBuilder
Elements	my \$W = new Win32::GUI::Window(); \$W->AddButton(...)	<Window> <Button .../> </Window>
Attributes	...->AddSomeWidget(-name => "SW", -text => "Go")	<SomeWidget name='SW' text=Go/>

Table 1: Mapping between Win32::GUI elements and Win32::GUI::XMLBuilder elements.

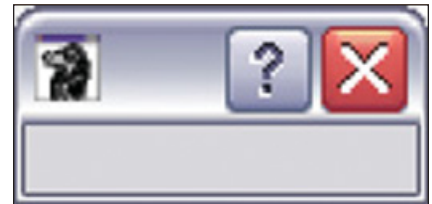


Figure 4: A dialog created with `<DialogBox width='100' height='50'/>`.

be evaluated. This is useful when you want to create dynamically sizing windows:

```
<GUI>
  <Window name='W' width='400' height='200'>
    <StatusBar name='S'
      text='status text'
      top='$self->{W}->ScaleHeight-$self->{S}->Height
        if defined $self->{S}'
    />
  />
</Window>
</GUI>
```

This shows how to create a simple Window with a *StatusBar* that automatically positions itself. The value of `$self->{W}->ScaleHeight-$self->{S}->Height if defined $self->{S}` is recalculated every time the user resizes the Window. (I'll come back to the topic of autoresizing later.)

Win32::GUI::XMLBuilder has an XML Schema

All WGX XML files will contain this basic structure:

```
<GUI>
  <Window/>
</GUI>
```

The `<GUI> .. </GUI>` (or in XML shorthand `<GUI/>`) elements are required and all WGX elements must be enclosed between these. In fact, WGX has a well-defined schema and we should really expand the above to the following XML:

```
<?xml version="1.0"?>
<GUI
  xmlns="http://www.numeninvest.com/Perl/WGX"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.numeninvest.com/Perl/WGX
    http://www.numeninvest.com/Perl/WGX/win32-gui-xml-
    builder.xsd">
  <Window/>
</GUI>
```

I chose not to, allowing me to simplify the code, but there is nothing stopping you from checking that your XML is well formed and valid using a tool such as Altova's XMLSPY (<http://www.altova.com/>). In fact, for large WGX programs, it is probably a good idea.

Top-Level Widgets

Every GUI application must have at least one top-level widget. These are the containers of your application. WGX currently supports several, including *Window*, *DialogBox*, and *MDIFrame*. They differ in behavior and functionality; for example, by default, a *DialogBox* cannot be resized and has no maximize or minimize buttons. See Figures 3 and 4.

Event Models

WGX supports two event models—New Event Model (NEM) and Old Event Model (OEM). NEM allows you to add attributes such as *onClick* or *onMouseOver*, containing anonymous subroutines to elements. The first argument to the subroutine is always the object in question. Our example:

```
<Button text='Push me!' onClick='sub
  {$_[0]->Text("Thanks")}' />
```

produces a *Button* widget with the initial text “Push me!” An anonymous *sub* is called when the user triggers the *onClick* event. This takes `$_[0]` (referring to the same *Button*) and applies the method *Text*, changing the text to “Thanks.”

Autoresizing

A big headache when writing Win32::GUI applications is managing *onResize* events. This is normally done by tredding through a top-level widget's children and explicitly specifying the dimensions that they should have if a *Resize* event occurs.

WGX will automatically generate an *onResize* NEM subroutine by reading-in the values for top, left, height, and width of each child. This will work sufficiently well, provided you use values that are dynamic, such as `$self->{PARENT}->Width` and `$self->{PARENT}->Height` for width and height attributes, respectively, when creating new widget elements. In fact, WGX will assume your child objects will have the same dimensions as their parents and will place them in the top left corner relative to their parent if you do not explicitly specify any dimensions.

In our simple button example, the *Button* element defaults to a position of *top* 0 and *left* 0 relative to its parent *Window*, with the exact same width and height as its parent window. To change this default, we must specify either *left*, *top*, *width*, or *height* attributes, or all these combined and separated with commas in a single *dim* attribute.

In Example 1, two widgets are specified. The *Label* widget only overrides the *height* attribute, thus taking on spatial defaults for

everything else. The *Button* attribute overrides *top* and *height* attributes. The *top* attribute is necessary to avoid overlapping the *Label* widget. Figures 5 and 6 show the resulting window in its initial state, and after resize.

You will have noticed the special value *%P%* being used in the aforementioned attributes. These are substituted by *WGX* for

WGX might be considered just another way to write Win32::GUI code, where Win32::GUI objects become XML elements and Win32::GUI options become XML attributes

\$self->{PARENT}, sparing us the need to name the top-level *Window* widget.

WGX Utility Elements

A common task when writing an application is building a simple GUI that can recall its last desktop location and can be maximized or minimized. Such GUI-specific tasks might not be integral to the rest of the application, and it can be helpful to keep those tasks isolated from the core code of the application. With *WGX*, one can embed GUI initialization and destruction code directly in the XML file. There are several pure *WGX* elements that come to aid here.

<WGXPre>, <WGXPost>, and <WGXExec>

These utility elements allow you to execute Perl code before, after, and during construction of your GUI. This allows interface-specific code to be embedded within your XML document:

```
<WGXPre><![CDATA[

    use Win32::TieRegistry(Delimiter=>"|",
                          ArrayValues=>0);
    our $registry = &initRegistry();

    sub initRegistry {
        ...
    }

    sub Exit {
        ...
    }

]]></WGXPre>
```

WGXExec and *WGXPre* elements are executed in place and before GUI construction. These elements can be useful when you need to implement something not directly supported by *WGX*, like *Toolbars*:

```
<Toolbar name='TB' height='10' />
<WGXExec><![CDATA[
```

```
$self->{TB}->AddString("File");
$self->{TB}->AddString("Edit");
$self->{TB}->SetStyle(TBSTYLE_FLAT
                    |TBSTYLE_TRANSPARENT
                    |CCS_NODIVIDER);

$self->{TB}->AddButtons(2,
    0, 0, TBSTATE_ENABLED, BTNS_SHOWTEXT, 0,
    0, 1, TBSTATE_ENABLED, BTNS_SHOWTEXT, 1,
);
]]></WGXExec>
```

Any output returned by *WGXExec* and *WGXPre* elements will be parsed as valid XML, so

```
<WGXExec><![CDATA[
    return "<Button text='Go' />"
]]></WGXData>
```

and

```
<Button text='Go' />
```

are equivalent. If the return data does not contain valid XML, then it will make your *WGX* document invalid—so check what these elements return, and if you are unsure, then add an explicit *return*.

WGXPost elements are executed after all GUI elements have been constructed and before your application starts its *Win32::GUI::Dialog* phase.

<WGXMenu>

This element can be used to create menu systems that can be nested many times

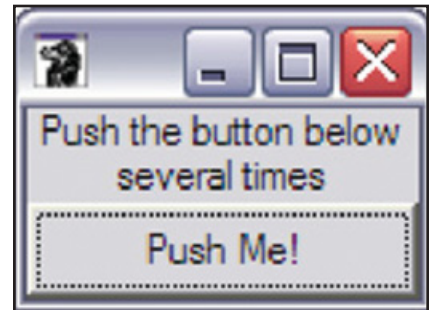


Figure 5: Window from Example 1 in its initial state.

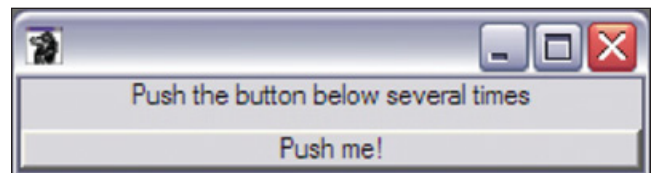


Figure 6: Window from Example 1 after resize.

```
<GUI>
  <Window height='90'>
    <Label
      text='Push the button below several times'
      height='%P%->ScaleHeight/2'
      align='center'
    />
    <Button
      text='Push me!'
      top='%P%->ScaleHeight/2'
      height='%P%->ScaleHeight/2'
      onClick='sub {
        $_[0]->Text($_[0]->Text eq "Thanks" ? "Push Me!"
          : "Thanks")
      }'
    />
  </Window>
</GUI>
```

Example 1: A window with *Label* and *Button* widgets overriding default values.

using the standard `Win32::GUI MakeMenu`. Although you can use `Item` elements throughout the structure, it is more readable to use the `Button` attribute when a new nest begins:

```
<WGXMenu name="M">
  <Button name='File' text='&File'>
    <Button text='New'>
```

*With WGX, one can embed GUI
initialization and destruction code
directly in the XML file*

```
<Item text='Type One Document' />
...
</Button>
<Item name='ExitPrompt'
  text='Prompt on Exit'
  checked='exec:$registry->{exitprompt}'
  onClick='sub {
    $self->{ExitPrompt}->Checked(not
      $self->{ExitPrompt}->Checked);
  }'
/>
<Item text='Exit' onClick='sub { Exit($_[0]) }' />
</Button>
<Button name='Help' text='&Help'>
  <Item text='Contents' />
  <Item separator='1' />
  <Item text='About' onClick='sub {
    Win32::GUI::MessageBox(
      $_[0],
      "... some info.",
      "About ...",
      MB_OK|MB_ICONASTERISK
    );
  }' />
</Button>
</WGXMenu>
```

You can see that separator lines can be specified by setting the `separator` attribute to 1. You will also notice menu items can also contain check boxes; in the earlier example, the check-box state is stored in the registry.

Example Listing

I hope I have given you a good base to start developing your own applications with `Win32::GUI::XMLBuilder`. I have provided a complete example with this article (available online at <http://www.tpj.com/source/>). It is the application template also available with the distribution, and should be a good starting point for any Win32 application.

TPJ

101 Perl Articles!



From the pages of *The Perl Journal*, *Dr. Dobb's Journal*, *Web Techniques*, *Webreview.com*, and *Byte.com*, we've brought together 101 articles written by the world's leading experts on Perl programming. Including everything from programming tricks and techniques, to utilities ranging from web site searching and embedding dynamic images, this unique collection of *101 Perl Articles* has something for every Perl programmer.

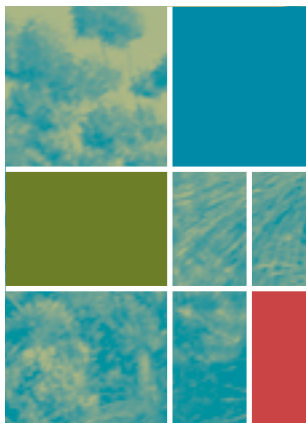
Plus, this collection of articles is fully searchable, and includes a cross-platform search engine so you can immediately find answers you're looking for. Delivered as HTML files in a ZIP archive or CD-ROM image, download *101 Perl Articles* and burn your own CD-ROM or store it on hard disk.

\$9.95 For subscribers to
The Perl Journal

\$12.95 For nonsubscribers to
The Perl Journal

\$24.95 To subscribe to
The Perl Journal and
receive *101 Perl Articles*

Go to
<http://www.tpj.com/>
now!



Using the Web as a GUI

Simon Cozens

I could never get on with GUI programming at all. I don't think very well in terms of the event loop paradigm. I don't want to spend a lot of time laying out widgets and connections, but I don't like the look of those toolkits like Tk that require you to pack widgets together. And to cap it all, I at least like to pretend that my applications are cross platform, and most GUI widget sets just aren't.

At the same time, I've been using HTML and CSS for pretty much everything—layout of documents for printing, presentation slides, you name it. And of course I'd been writing lots of web applications with Maypole. Why shouldn't I use a web browser to provide the GUI to a nominally web-based application instead of writing a true GUI program?

Of course, it's hardly a new idea. Activestate's Komodo is an example of a sophisticated application based on top of the Mozilla browser platform. I recently had to write a Windows application, but develop it on the Mac, so I chose to write it using HTML and CSS for the display, Javascript for the client side, and a Maypole back-end to connect the whole thing to a database. The application runs in a web browser, using a local web server, but the end user doesn't need to know or care—from their point of view, a window pops up on the screen and they interact with it.

This, of course, requires a local web server, a copy of Perl, and almost half of CPAN, some things that Windows is notorious for not providing. Additionally, we don't really want the user to go through a laborious process of installing and setting up all these complicated systems. Ideally, we want the single executable to do everything itself, with no installation required. To make this happen, we're going to have to write our own web server and package it all up—the server, the application, the browser, the templates to be displayed, and everything else—into a single binary. Let's first look at the web server.

The Web Server

I'll start by saying that none of the ideas that I've used in this article are original; we all stand on the shoulders of giants. There are many modules and methods for creating a web server in Perl, but I've used the *standalone_httpd* from the RT web application (<http://www.bestpractical.com/rt/>). RT is trying to do the same

Simon is a freelance programmer and author, whose titles include Beginning Perl (Wrox Press, 2000) and Extending and Embedding Perl (Manning Publications, 2002). He's the creator of over 30 CPAN modules and a former Parrot pumpking. Simon can be reached at simon-cozens.org.

sort of thing that we're doing—having a web server that only knows how to talk to the RT application so that it can all be bundled into a single program.

standalone_httpd is a simply designed server, with the emphasis on portability and speed. Let's take a look at how it's constructed and how we adapt it for our Maypole application. We'll be talking about Maypole for our purposes, but similar considerations would be applicable to any situation where you're trying to build a compact web server around an application.

First, we use the old-fashioned Socket operations to bind to the web server port and listen for connections. It may be ugly, but it's fast, and that's what counts here. We're looking for a real-time response, just like you'd get with a conventional GUI application, without the overhead of making HTTP connections, so we need to cut down as much extraneous stuff as possible.

```
my $port = shift;
my $tcp = getprotobyname('tcp');

socket( HTTPDaemon, PF_INET, SOCK_STREAM, $tcp )
    or die "socket: $!";
setsockopt( HTTPDaemon, SOL_SOCKET, SO_REUSEADDR,
    pack( "l", 1 ) ) or warn "setsockopt: $!";
bind( HTTPDaemon, sockaddr_in( $port, INADDR_ANY ) )
    or die "bind: $!";
listen( HTTPDaemon, SOMAXCONN ) or die "listen: $!";

print("You can connect to your RT server at
http://localhost:$port/\n");
```

Now that we've set up the listening socket, we can take requests:

```
while (1) {

    for ( ; accept( Remote, HTTPDaemon );
        close Remote ) {

        *STDIN = *Remote;
        *STDOUT = *Remote;
        chomp( $_ = <STDIN> );
```

We accept the remote socket, and then set up standard input and standard output to read from and print to that, respectively; this mimics the usual CGI environment. We also read the first line of the HTTP request from the socket. Again, we could use

HTTP::Request to do this, but we need to keep it lean and lightweight.

From this line of the request, we can read off the method, the URI, any GET parameters, and check that we're looking at a valid request:

```
my ( $method, $request_uri, $proto, undef ) = split;

my ( $file, undef, $query_string ) =
  ( $request_uri =~ /(^[^?]*)(\?(.*)?)?/ ); # split at ?

last if ( $method !~ /^(GET|POST|HEAD)$/ );
```

We're going to have to write our own web server and package it all up—the server, the application, the browser, the templates to be displayed, and everything else—into a single binary

Next, we dispatch to a function that turns all of these things into the kind of CGI environment variables that we would expect:

```
build_cgi_env( method      => $method,
               query_string => $query_string,
               path         => $file,
               method       => $method,
               port         => $port,
               peername     => "localhost",
               peeraddr     => "127.0.0.1",
               localname    => "localhost",
               request_uri  => $request_uri );
```

We won't go into all the details of how that does its job, but we should know that at this point, our program looks very much like an ordinary CGI script. So it shouldn't be much of a surprise that the RT standalone HTTP server now just creates a CGI object and runs it through its *HTML::Mason* handler, which does all the processing and spits out the output to the client:

```
RT::ConnectToDatabase();
my $cgi = CGI->new();
print "HTTP/1.0 200 OK\n"; # probably OK by now
eval { $h->handle_cgi_object($cgi); };
```

And that's basically it—a web server that contains everything it needs to respond to a request and hand it over to RT. Now we want to modify this so that instead of running an *HTML::Mason* handler, it runs our Maypole application.

Adjustments for Maypole

We wrap this program up into *Maypole::HTTPD* and customize the part that responds to the CGI request. Maypole already has a CGI driver, *CGI::Maypole*, so it's reasonable to use that. However, Maypole uses *CGI::Simple*, and it turns out for some rea-

son that *CGI::Simple* doesn't like our CGI environment; additionally, the RT server always returns 200 OK, but we might not want to do that on some occasions. Finally, a Mason request will automatically handle static files that need to be served from the application, such as logos, CSS and XSL files, and so on, but we don't have code in Maypole to handle this, so we need to be able to serve files as well as pass things through the Maypole process. Thankfully, in the application I had, I knew that every URL containing `/static/` related to a static file that we needed to serve up.

So we'll begin by laying out the things our code will need to do—serve a file, or pass the request to Maypole and send the output:

```
if ($path =~ /static/) { return $self->serve($path) }

print "HTTP/1.1 200 OK\n";
# Do something Maypole here
```

Let's deal with serving files, which is the normal use of a web server but rather incidental to what we're doing. With *serve*, we're given a path, and we need to turn this into a file and serve it up with the correct MIME type:

```
use File::Spec::Functions qw(canonicalpath);
use File::MMagic;
use URI::Escape;

sub serve {
  my ($self, $path) = @_;
  $path = ".".canonicalpath(uri_unescape($path));
  if (-e $path and open FILE, $path) {
    binmode FILE;
    print "HTTP/1.1 200 OK\n";
    my $magic = File::MMagic->new();
    print "Content-type: ",
          $magic->checktype_filename($path), "\n\n";
    print <FILE>;
    return;
  }
  print "HTTP/1.1 400 Not found\n";
}
```

We're using three common CPAN modules here: *File::Spec::Functions* is not only used to handle filenames in a platform-agnostic way, its *canonicalpath* function allows us to stop any file access attacks—if the user looks for `http://localhost/../../../../etc/passwd`, then we need to stop that. *canonicalpath* treats the path as being absolute, so it strips out the initial `../`s, leaving us with `/etc/passwd` that's, we hope, won't be found.

URI::Escape allows us to convert the filenames from their encoded form—with `%20` for space, for instance—to the form that the filenames would take on the disk. If after these two measures, we can open a filehandle, then we have a file to serve and we can finally send the OK status code.

At this point, we need to know what MIME type to send to the browser so that the file can be displayed properly; a PNG file to be used as a logo, for instance, needs to be served with type `image/png`. The *File::MMagic* module sniffs the first few bytes of a filehandle and determines the appropriate MIME type to send. Then we can send the payload of the file, and all is fine.

Next is the more common case of processing a request through Maypole. To make this happen, we need to know in our main loop the name of the Maypole application to call, we need to ensure it's based on *CGI::Maypole*, and then we can use the handy *run* method to process the request, much like Mason's *handle_cgi_object*. So we modify our *main_loop* to take an application name as well as a port:

```
sub main_loop {
    my ($self, $module, $port) = @_;
    $port ||= 8080;
```

Next, we check that the application is loaded, and then fiddle it so that it's based on *CGI::Maypole*:

```
$module->require;
{ no strict;
    local *isa = *{$module."::ISA"};
    unshift @isa, "CGI::Maypole"
        unless $isa[0] eq "CGI::Maypole"
}
```

PAR stands for Perl ARchive and is a Perl analogue of Java's JAR system—essentially a Zip file of a program and everything that Perl needs to run it

Finally, when we come to handle the request, we just need to say

```
$module->run;
```

and we have a working server.

The Client

To give the impression that this is not a client-server application but a standard GUI application, we need to write a wrapper program that starts up the server, starts up a web browser, and points it at the right address. This is where we need to be slightly platform specific, but thankfully the driver script is very short. Here's the driver for the application I was writing, called "Songbee":

```
use Songbee;
use Maypole::HTTPD;

$x = fork or Maypole::HTTPD->main_loop("Songbee");
system("firefox http://localhost:8080/");
kill 1, $x, $$;
```

This works well enough on both Windows and UNIX; it forks a process to run the web server part, and then runs the web browser. When the web browser is done, it kills both processes. It needs to do this because on Activestate windows, the "forked" process isn't really forked, it's just a thread of the main process, so we need to kill \$\$.

Now we come to the most difficult bit—working out how to package together all these elements, plus all the associated data, into a single file.

PARring the Code Together

This is where Autrijus Tang's "PAR" comes in. PAR stands for Perl ARchive, and is a Perl analogue of Java's JAR system—essentially a Zip file of a program and everything that Perl needs to run it.

At its very simplest, PAR is just a mechanism that allows you to read modules from inside a zip file. Once you've created the zip file, like so:

```
% zip modules.par lib/Songbee.pm lib/Songbee/HTTPD.pm ...
```

you can use the PAR module to treat it as an include path:

```
use PAR;
use lib "modules.par"; # Now we can find
                        # Songbee and friends
use Songbee::HTTPD;
```

Of course, just loading Songbee.pm and the other files is no good if you don't have the modules that they depend on. Thankfully, there's a very helpful tool called *Module::ScanDeps* that reports on the dependencies of a given Perl program. So running it on the driver that we wrote earlier, we get a whole raft of dependencies that are going to need to go into our PAR when we run the program on a "clean" Windows computer without Perl installed:

```
% scandeps.pl songbee.pl
'Class::DBI::Loader'      => '0.02',
'Songbee'                 => 'undef',
'Songbee::HTTPD'          => 'undef',
'Compress::Zlib'          => '1.32',
'CGI::Simple'             => '0.075',
'Maypole'                 => '1.5',
'CGI::Simple::Cookie'     => '0.02',
'CGI::Simple::Util'       => '0.002',
'Class::DBI::ColumnGrouper' => 'undef',
'Class::Data::Inheritable' => '0.02',
...
```

Now all we need to do is put these things together—the driver, the archive of the modules, the automated dependency scanning—so that we run one command and end up with an archive that contains the program and everything we need to run it. Thankfully, PAR does that, too.

PAR comes with a binary called the "Perl Packager" (pp). This does everything that we need, such that we can say:

```
% pp -a -o songbee.par songbee.pl
```

This will create songbee.par from songbee.pl and all its dependent Perl modules. Now we can use the PAR Loader, parl, to run this:

```
% parl songbee.par
```

and we find that...it doesn't work. Unfortunately, pp only statically analyzes the program for modules that are used or required; it knows nothing about modules that are required dynamically. For instance, Songbee uses SQLite as its database, but this is only determined at runtime—nowhere is there an explicit *use DBD::SQLite*, so the module is not picked up by pp. We can provide a list of additional modules for pp to pick up by mentioning them on the command line:

```
% pp -a -o songbee.par -MDBD::SQLite -M... songbee.pl
```

But since there are a lot of them, I found it easier just to add explicit use statements to the driver:

```
use DBD::SQLite;
use DBIx::ContextualFetch;
```

```

use Class::DBI::Loader;
use Class::DBI::Loader::SQLite;
use Class::DBI::SQLite;
use Class::DBI::Relationship::HasA;
use Class::DBI::Relationship::HasMany;
use Maypole::Model::CDBI;
use Maypole::View::TT;
use Template::Plugin::XSLT;

```

Now everything works. Well, sort of. That last line, *use Template::Plugin::XSLT*, also pulls in *XML::LibXML* and *XML::LibXSLT*, and they in turn require some dynamically loaded C libraries to be available.

This is no problem for pp, so long as we inform it, and we can use the *-l* switch to point it at the libraries in question:

```

pp -a -l c:\perl\bin\libxml2.dll -l
c:\perl\bin\libxslt_win32.dll -l
c:\perl\bin\libxslt_win32.dll -o songbee.par song-
bee.pl

```

(It was at this point that I switched to a batch file to construct my PAR files.)

Now we've gotten rid of most of the dependencies into the one PAR file: What remains outside are the templates, the browser, and, of course, Perl itself. Thankfully, the last bit is easy to get rid of—by dropping the *-a* option, pp will no longer simply produce a .par file but will also bundle up the Perl interpreter with it and produce a standalone executable:

```

pp -l c:\perl\bin\libxml2.dll -l
c:\perl\bin\libxslt_win32.dll -l
c:\perl\bin\libxslt_win32.dll -o songbee.exe song-
bee.pl

```

We run this program, the browser window pops up, the templates are loaded and work, and the end user just sees an application on his screen. All is well.

Now the final piece of the puzzle is to hide all the data inside the .exe as well.

PARring Data

PAR provides us with a way of packaging up files, and indeed, entire directories inside our PAR Zip files, as well as the Perl modules that live in there. When a PAR-based application runs, PAR extracts the contents of the Zip file to a temporary directory. It then provides a hook into the *@INC* mechanism so that module files can be found via the temporary directory. Additionally, it puts the name of the temporary directory in the environment variable *PAR_TEMP*, and provides the subroutine *PAR::read_file* to read a data file from the archive.

So the first problem is getting all the data files into the archive. I did this by creating a manifest file, like so:

```

static
custom
factory
playitem
playlist
song
workshop.db
firefox.exe
...

```

I could then feed this to pp with the *-A* parameter. Most of the entries in this file are directories, but pp includes all the files in them recursively.

Now we have a significantly larger PAR file, but we're not using the data in it yet. To do this, we could fix our application to use *PAR::read_file* every time it wants to open a data file, but this is pretty difficult—as well as rewriting the part of the web server that serves up static files, we'd have to reach into the bits of Maypole that look for templates.

A much easier way is to simply change to the directory that all the data is in. We add this to our driver:

```

$ENV{PAR_TEMP} && chdir($ENV{PAR_TEMP});

```

And of course, everything will work without further modification.

The Proof of the Pudding

Now we can serve up files, start the Firefox browser, and everything else, in the right place—with all of the code and data coming out of the single .exe file produced by pp.

As a test—and since this is exactly what I need to do when I deploy the program—I sent the executable to a friend who I knew didn't have Perl, Firefox, or anything else installed; he double-clicked the nice icon, and up popped a window. No messy installation, tedious set-up, or anything.

By using HTML elements as the GUI, I've saved myself a lot of bother with GUI programming and have been able to use Maypole to get the application coded quickly. And by using this client-server mechanism, I've been able to develop on Macintosh, run on Linux, and ship to friends on Windows.

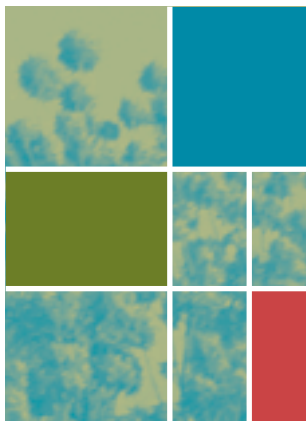
TPJ

**Fame & Fortune
Await You!**

**Become a
TPJ
author!**

The Perl Journal is on the hunt for articles about interesting and unique applications of Perl (and other lightweight languages), updates on the Perl community, book reviews, programming tips, and more.

If you'd like to share your Perl coding tips and techniques with your fellow programmers—not to mention becoming rich and famous in the process—contact Kevin Carlson at kcarlson@tpj.com.



Scripts as Modules

brian d foy

Later, I've been writing scripts that look more like modules. Some of this comes naturally from writing so many modules, but I've also discovered that these scripts are easier to manage and test. The result is a Perl file that acts like a module when I use it like a module, and acts like a script when I use it like a script.

When I first started writing Perl, I mostly wrote the usual type of script: I started at the top of the page and wrote statements that I expected to execute in order as I moved down the page. I could read it like a movie script, going from one line to the next.

Later, as I learned more Perl and got more programming experience (even with other languages such as C, Java, and Smalltalk), I started using more functions. The code, however, was still very procedural and I could follow it linearly down the page, even if I had to look at a function definition every so often.

In the past couple of years, I've gotten on the testing bandwagon. I want to test everything. The *Test::More* module made that very easy, and there has been an explosion in the number of *Test::** modules to check various things (and I'm responsible for a couple of them).

Scripts are hard to test, though. I can choose the input at the start, such as the environment variables, command-line arguments, and standard input. After that I have to wait for the script's output to see if things went right. If things didn't go right, I have to figure out where, between the first line and the last line, things went wrong.

A module is relatively easy to test. A good module author breaks down everything into methods (or plain old functions) that do one task or small bits of a task. As long as the methods don't use side effects (like looking at global variables, including nonlocal versions of Perl's special variables), I can give each method some arguments and check its return values. Once I test each of the methods, I can confidently use them knowing that they do what I expect. When things go wrong, I have a lot less to search through to find the problem.

The Core Structure

To create this sort of script, which I have been calling a "modulino," I take Perl back a step. Remember the *main()* function of a

brian has been a Perl user since 1994. He is founder of the first Perl Users Group, NY.pm, and Perl Mongers, the Perl advocacy organization. He has been teaching Perl through Stonehenge Consulting for the past five years, and has been a featured speaker at The Perl Conference, Perl University, YAPC, COMDEX, and Builder.com. Contact brian at comdog@panix.com.

C program, and how Perl made the whole script file *main()* and called it *main::*? I need that back, but I'm going to call it the *run()* method:

```
#!/usr/bin/perl
package Local::Modulino;

__PACKAGE__->run( @ARGV ) unless caller();

sub run { print "I'm a script!\n" }

__END__
```

My modulino no longer assumes that it is in the *main::* namespace and that the whole file is the script. I need to put everything that I want to do in the *run()* method, just like I would do with C's *main()*. As a start, my modulino just prints a short message.

The script-or-module magic works in the third line, which checks the result of the *caller()* Perl built-in function. If something else in the Perl script calls the file, the *caller()* function (in scalar context) returns the calling package name. That's true if another Perl file loads this one with *use()* or *require()*. If I run my modulino as a script, there is no other file loading it, so *caller()* returns *undef*. If *caller()* returns a false value, then I execute the *__PACKAGE__->run()* method (the Perl compiler replaces the *__PACKAGE__* token with the current package name).

That's it. That's the core of the dual-duty Perl modulino. Everything else is just programming.

I save this file as *Modulino.pm* and execute it in a couple of different ways. From the command line, I can call it like a script. The *caller()* expression returns *False*, so Perl executes the *run()*, which prints out my short message:

```
prompt% perl Modulino.pm
I'm a script!
prompt%
```

When I load the file as a module using the *-M* switch (which works like *use()*), the *caller()* expression returns a true value (extra credit for knowing what the actual value is), so *unless()* doesn't evaluate the rest of the statement. The *run()* never runs, and I don't get any output:

```
prompt% perl -MModulino.pm -e 1
prompt%
```

I can still get output though—I just need to call the `run()` method myself:

```
prompt% perl -MModulino -e 'Local::Modulino->run()';
I'm a script!
prompt%
```

The script-or-module magic works in the third line, which checks the result of the caller() Perl built-in function

The Rest of the Story

Now that I have the basic structure of the modulino, I need to apply it to something useful, but that's probably too long for the space I have left in this article. Well, maybe not. For a while, I've wanted a little tool to download the RSS feed from *The Perl Journal* and print a table of contents. Unlike the PDF files I get for each issue, I always know where the RSS file is: It's the same URL every time.

I wrote a modulino to download, parse, and display the table of contents of *The Perl Journal*. I could have written this as a script and gone through each of those steps in sequence, but with a modulino, I have a bit more flexibility; and when I decide to test it, I should be able to find problems easier and faster.

In `Local::Modulino::RSS` (see Listing 1), I display the table of contents as text, which works just fine for me in my terminal window. However, since I structured the code as a module, I could very easily do something else. Perhaps I want to convert the table of contents into HTML so I can display it on my personal home page. Since only my `run()` knows anything about the data presentation, I just have to override it, which I show later.

The rest of the modulino is a collection of very short functions doing a very specific task. I can easily write some testing code to make sure each of the small functions does what I think they should. I skip that part here since the topic has been covered so well in other articles.

On line 1, I start with a shebang line. If I want to run this as a Perl script without specifying “perl” on the command line, the operating system needs to know which interpreter I intend to use.

Next, I define the package name and invoke the `run()` method if I call the file as a script. If I use this file as a module, `caller()` returns true and I don't call the `run()` method.

On line 8, I define my `run()` method. On line 11, I take the first argument, which is the package name, off of the argument list. Each method does this, so I can subclass the task. I call each function as a class method so inheritance works out right. The methods will always know who is calling them, even if it is a derived package.

Most of the complexity of the task is hidden behind functions. The `fetch_items()` method is composed of the `get_url()`, `get_data()`, and `get_items()` methods that do most of the actual work. My `run()` method simply gets the parts it needs. This way, when I want to write another `run()` method, I won't have to do so much work.

On line 13, I go through each item and extract the information for that issue. The `get_issue()` function returns the title of the item, which turns out to be the month and year of publication, along with the articles in that issue as a list of *article title*, *author* anonymous array pairs. It's the data, so up to this point, I can still do just about anything I like, but once I have the text for the title and the articles for the latest issue, I simply print them to the terminal as plain text, as I show in Listing 2.

Some of you might have noticed the start of a model-view-controller (MVC) design (although I don't have much controller going on). The data handling and the presentation don't depend on each other. The MVC design, which may sound fancy or exotic, naturally pops up when I use a lot of small functions to do single tasks. The only part of my script that deals with the presentation of the data is the `run()` method, and that's easy to override with a subclass.

In fact, it's so easy to subclass that I might as well do it here. In Listing 3, I create the `Local::Modulino::RSS::HTML` modulino, although it only overrides the `run()` method by defining its own version. I have to tell it that it is a subclass of `Local::Modulino::RSS` with the `use base` declaration so it looks in that class for methods it does not define, such as `fetch_items()` and `get_issue()`. I also *require* “RSS.pm” because I didn't bother to install these files as proper modules, so I don't want my modulino to look in `Local/Modulino/RSS.pm` to find the file. I show the new output format in Listing 4.

Code

By creating a modulino, I get my Perl scripts to do double duty as scripts and as modules. If I structure the code as a module, I can reuse and override it just like a module. Since I broke everything down to small functions instead of using a procedural style, I also make things easier to test.

TPJ

(Listings are also available online at <http://www.tpj.com/source/>.)

Listing 1

```
1 #!/usr/bin/perl
2 package Local::Modulino::RSS;
3
4 __PACKAGE__->run() unless caller();
5
6 use HTML::Entities;
7 use Data::Dumper;
8
9 sub run
10 {
11     my $class = shift;
```

```
12
13     foreach my $item ( $class->fetch_items )
14     {
15         my( $title, @articles ) = $class->get_issue( $item );
16
17         print "\n$title\n-----\n";
18         printf "%-45s %-30s\n", @$_ foreach ( @articles );
19     }
20
21 }
22
23 sub fetch_items
24 {
25     my $class = shift;
26
27     my $url = $class->get_url();
```

```

28 my $data      = $class->get_data( $url );
29 my @items     = $class->get_items( $$data );
30 }
31
32 sub get_issue
33 {
34     my $class    = shift;
35
36     my $title    = $class->get_title( $_[0] );
37     my @articles = $class->get_articles( $_[0] );
38
39     return ( $title, @articles );
40 }
41
42 sub get_articles
43 {
44     my $class    = shift;
45
46     my $d = $class->get_description( $_[0] );
47
48     my @b = split /<br>s*<br>/, $d;
49     my @articles = ();
50
51     foreach my $b ( @b )
52     {
53         my @bits = split /<br>/, $b;
54         $author = pop @bits;
55
56         my $title = join " ", @bits;
57
58         $class->_normalize( $author, $title );
59         push @articles, [ $title, $author ];
60     }
61
62     @articles;
63 }
64
65 sub get_description { $_[0]->_field( $_[1], 'description' ) }
66 sub get_title       { $_[0]->_field( $_[1], 'title'       ) }
67 sub get_items       { $_[0]->_field( $_[1], 'item'       ) }
68
69 sub _normalize
70 {
71     my $class    = shift;
72
73     foreach ( 0 .. $#_ )
74     {
75         $_[$_] =~ s/^\s+|\s*$//g;
76         $_[$_] =~ s|</?>||g;
77         $_[$_] =~ s|\s+| |g;
78     }
79 }
80
81 sub _field
82 {
83     my $data = $_[1];
84
85     HTML::Entities::decode_entities( $data );
86
87     my @matches = $data =~ m|<Q$_[2]\E>(.*?)</Q$_[2]\E>|sig;
88
89     wantarray ? @matches : $matches[0];
90 }
91
92 sub get_data
93 {
94     my $class    = shift;
95
96     require LWP::Simple;
97     my $data = LWP::Simple::get( $_[0] );
98     defined $data ? \ $data : $data;
99 }
100
101 sub get_url {
102     "http://syndication.sdmediagroup.com/" .
103     "feeds/public/the_perl_journal.xml"
104 }

```

Listing 2

September 2004 PDF

Objective Perl: Objective-C-Style Syntax And Runtime for Perl Kyle Dawkins
 Scoping: Letting Perl Do the Work for You David Oswald

Secure Your Code With Taint Checking
 Detaching Attachments
 Unicode in Perl
 PLUS Letter from the Editor Perl News

Andy Lester
 brian d foy
 Simon Cozens
 Source Code Appendix

August 2004 PDF

Regex Arcana
 XML Subversion
 OSCON 2004 Round-Up
 Molecular Biology in Perl
 Pipelines and E-mail Addresses
 PLUS Letter from the Editor Perl News

Jeff Pinyan
 Curtis Lee Fulton
 Andy Lester
 Simon Cozens
 brian d foy
 Source Code Appendix

Listing 3

```

#!/usr/bin/perl
package Local::Modulino::RSS::HTML;

use base qw( Local::Modulino::RSS );

require "RSS.pm";

__PACKAGE__->run() unless caller();

use HTML::Entities;

sub run
{
    my $class    = shift;

    foreach my $item ( $class->fetch_items )
    {
        my( $title, @articles ) = $class->get_issue( $item );

        print "\n<h3>$title</h3>\n\n<ul>\n";
        printf "<li><b>%s</b>", %s\n", @$_ foreach ( @articles );
        print "</ul>\n";
    }
}

```

Listing 4

<h3>September 2004 PDF</h3>

 Objective Perl: Objective-C-Style Syntax And Runtime for Perl,&br/>
 Kyle Dawkins
 Scoping: Letting Perl Do the Work for You,&br/>
 David Oswald
 Secure Your Code With Taint Checking,&br/>
 Andy Lester
 Detaching Attachments,&br/>
 brian d foy
 Unicode in Perl,&br/>
 Simon Cozens
 PLUS Letter from the Editor Perl News,&br/>
 Source Code Appendix

<h3>August 2004 PDF</h3>

 Regex Arcana,&br/>
 Jeff Pinyan
 XML Subversion,&br/>
 Curtis Lee Fulton
 OSCON 2004 Round-Up,&br/>
 Andy Lester
 Molecular Biology in Perl,&br/>
 Simon Cozens
 Pipelines and E-mail Addresses,&br/>
 brian d foy
 PLUS Letter from the Editor Perl News,&br/>
 Source Code Appendix

TPJ



The Perl CD Bookshelf

Version 4.0

Jack J. Woehr

It's to be strenuously doubted whether any reader of *The Perl Journal* needs to be told why the Perl programming language is interesting and important. On the other hand, it's certain that many readers are in the midst of the learning process. These readers may well wish to consult *The Perl CD Bookshelf*, a new version of which, 4.0, has just been released by O'Reilly.

The Perl CD Bookshelf is pretty much a must-have for any Perl team's office library. It consists of the complete text of six famous and worthy Perl books attractively and economically laid out for onscreen reading in a web browser. In addition to appearing on the CD-ROM, one of the texts is also included as a trade glossy paperback in the package. There's also a platform-independent Java search engine. The *Bookshelf* sets a high standard for presenting programming literature in machine-readable form.

On the grounds that life is not all beer and skittles, I beg to present a couple of trivial quibble points:

1. There's a bit of Win-centrism. The search engine exhibits some minor misbehavior under UNIX browsers, and the matrix of browser compatibilities is presented as (guess what?) an Excel spreadsheet.
2. The various archives of code samples and examples, which one normally downloads from O'Reilly's web site upon purchasing a book, are not included on the CD-ROM. While the examples are indeed all present inline in the hypertext of the books themselves, and there's nothing stopping me from cutting and pasting them, I get the warm fuzzies from having all my sample code in plaintext in a .zip or .tar.gz archive. It's not like the CD-ROM was full.

Now let's look at each of the six books that make up the package:

***Perl in a Nutshell*, 2nd Edition**

Nathan Patwardhan, Ellen Siever, and Stephen Spainhour
ISBN 0596002416

Published June 2002

The O'Reilly "Nutshell" series was conceived to impart minimally the core of various programming and technical competencies without being terse or cryptic. *Perl in a Nutshell* is a coconut shell,

Jack J. Woehr is an independent consultant and team mentor practicing in Colorado. He can be contacted at <http://www.softwoehr.com/>.

***The Perl CD Bookshelf* Version 4.0**

O'Reilly Press, 2004
600 pp., \$99.95
ISBN 0596006225

weighing in at about 740 pages, making it probably the biggest "Nutshell" book ever. That reflects Perl's size and weight: Perl has always tended to shove features inwards towards the core, while the language's maturity and cheerfully insouciant willingness to embrace change have deposited layered strata of syntax. This toe-breaker is the one book included in the package in hard-copy. It's a good choice, as it is the most comprehensive instructional reference to the language yet printed.

***Perl Cookbook*, 2nd Edition**

Tom Christiansen and Nathan Torkington

ISBN 0596003137

Published August 2003

Yes, it's just like a cookbook. This massive "how-to" book (over 930 pages in print) is a nicely organized selection of examples and explanations of how to do the obvious; the sorts of things that aren't obvious sometimes when you're in a hurry. Need a friendly suggestion from the masters of the Perl arcana? You'll find it here.

***Programming Perl*, 3rd Edition**

Larry Wall, Tom Christiansen, and Jon Orwant

ISBN 0596000278

Published July 2000

Known in the industry as "the camel book," this (or its predecessor editions) is the book from which most of us learned Perl. The "pink camel book" covered early Perl while the 2nd edition, the "blue camel

book,” added discussion of object Perl, which had appeared in the interim. In a discipline already renowned for witty literary efforts, *Learning Perl* raised the bar, due largely to the manic evangelistic instructional style of Perl creator Larry Wall’s original coauthor Randal Schwartz. Less encyclopedic than *Perl in a Nutshell* but complete (up to Perl 5.6) and eminently readable, this is probably the rank beginner’s best introduction to the entire scope of the Perl language, the pre-6.0 Perl environment, and to the Tao of Perl. Schwartz later moved over to the *Learning Perl* authorship team. Much of Schwartz’s more memorable prose seems to have vanished from subsequent editions of *Programming Perl*, but despite the resultant increase in the sobriety of the presentation, the value of the treatise lies in its comprehensiveness, authority, pace of learning and readability, all of which remain undiminished over the intervening time.

***Learning Perl*, 3rd Edition**
Randal L. Schwartz and Tom Phoenix
ISBN 0596001320
Published July 2001

This is the “llama book,” wherein Schwartz and his talented co-author teach basic Perl in a fashion more amenable to quick learning of the core language than that provided in either of the other two introductory books in this bookshelf. Perl references and objects are banished to the appendices, and precious little is said of them even there. If you want a quick bootstrap into Perl’s classic use as a text-processing language, this is the corner of the bookshelf you should visit first.

Learning Perl Objects, References & Modules
Randal L. Schwartz
ISBN 0596004788
Published June 2003

Okay, so they left object Perl out of *Learning Perl*. Here Schwartz makes up the difference. As the book commences:

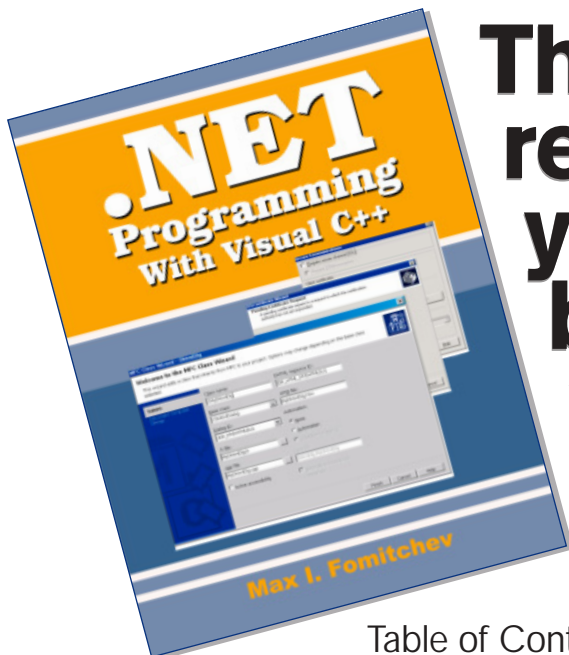
“Welcome to next step in your understanding of Perl. You’re probably here either because you want to learn to write programs that are more than 100 lines long or because your boss has told you to do so.”

That gives you an idea of the flavor of Schwartz’s approach; i.e., he’s both funny and right on. Perl very quickly does become unmanageable without objects: With objects, it’s merely nearly unmanageable. As with *Learning Perl*, this book is the way to proceed if you want the quick bootstrap, this time into Perl objects, largely because the humor and sympathy of the author make the task less onerous.

***Mastering Regular Expressions*, 2nd Edition**
Jeffrey Friedl
ISBN 0596002726
Published July 2002

Over the past 20 years, I’ve come to certain conclusions about software engineering. One is that “all applications are database applications,” and another is that “all text processing applications are regex applications.” Regular expressions have grown with time from being a peculiar feature of Unix to being part of every programmer’s practice, especially in Perl, wherein regexes assume almost supernatural power. O’Reilly calls this meticulous monograph a “bonus book” and presents it in Adobe PDF format. The author maintains his own website for this book at <http://www.regex.info/>.

TPJ



The .NET resource you’ve been waiting for!



- Delivered in PDF format.
- Packed with C++ code examples.
- Thousands of lines of source code.
- A complete reference to the .NET Framework

Table of Contents and sample chapter available at:
<http://www.ddj.com/dotnetbook/>

Get your copy now! Available via **download** for just **\$19.95**
 or
 on **CD-ROM** for only **\$24.95** (plus s/h).

Source Code Appendix

John Graham-Cumming “Practical Secure Port Knocking”

Listing 1

```
#!/usr/bin/perl -w
use strict;
my @knocks;
while ( <> ) {
    my $packet = $_;
    $packet =~ /^((\d+\.){3}\d+)\.(\d+) > ((\d+\.){3}\d+)\.(\d+)/;
    my ($src_ip,$src_port,$dest_ip,$dest_port) = ($1,$3,$4,$6);
    if ( $packet =~ / ack / ) {
        @knocks = grep( !/^$src_ip:$src_port$/, @knocks );
    } else {
        push @knocks, "$dest_ip:$dest_port";
    }
}
foreach my $k (@knocks) {
    print "Knock on $k\n";
}
```

Listing 2

```
# The common section contains configuration options for the tumblerd daemon,
# here we set the UDP port to listen on to 8675 and a log file
[common]
port = 8675
log = /var/log/tumblerd.log
# Each door that a user can knock on is defined by a unique [door-X] section,
# the first section is for opening the SSH port, and second for closing
#
# Each door has a secret (i.e. the password for this
# door that is part of the knock) and a command to execute.
#
# In the command it's possible to use the macros %IP% for the IP address of
# the person who knocked and %NAME% for the name of the door (in the
# first door here the name is open-ssh)
[door-open-ssh]
secret = open-pAsSwOrD
command = iptables -I INPUT -p tcp -s %IP% --dport 22 -j ACCEPT
[door-close-ssh]
secret = close-pAsSwOrD
command = iptables -D INPUT -p tcp -s %IP% --dport 22 -j ACCEPT
```

brian d foy “Scripts as Modules”

Listing 1

```
1  #!/usr/bin/perl
2  package Local::Modulino::RSS;
3
4  __PACKAGE__->run() unless caller();
5
6  use HTML::Entities;
7  use Data::Dumper;
8
9  sub run
10 {
11     my $class = shift;
12
13     foreach my $item ( $class->fetch_items )
14     {
15         my( $title, @articles ) = $class->get_issue( $item );
16
17         print "\n$title\n-----\n";
18         printf "%-45s %-30s\n", @$_ foreach ( @articles );
19     }
20
21 }
22
23 sub fetch_items
24 {
25     my $class = shift;
26
27     my $url = $class->get_url();
28     my $data = $class->get_data( $url );
29     my @items = $class->get_items( $$data );
30 }
31
32 sub get_issue
33 {
34     my $class = shift;
35
```

```

36 my $title    = $class->get_title( $_[0] );
37 my @articles = $class->get_articles( $_[0] );
38
39 return ( $title, @articles );
40 }
41
42 sub get_articles
43 {
44     my $class    = shift;
45
46     my $d = $class->get_description( $_[0] );
47
48     my @b = split /<br>\s*<br>/, $d;
49     my @articles = ();
50
51     foreach my $b ( @b )
52     {
53         my @bits = split /<br>/, $b;
54         $author = pop @bits;
55
56         my $title = join " ", @bits;
57
58         $class->_normalize( $author, $title );
59         push @articles, [ $title, $author ];
60     }
61
62     @articles;
63 }
64
65 sub get_description { $_[0]->_field( $_[1], 'description' ) }
66 sub get_title       { $_[0]->_field( $_[1], 'title'       ) }
67 sub get_items       { $_[0]->_field( $_[1], 'item'       ) }
68
69 sub _normalize
70 {
71     my $class    = shift;
72
73     foreach ( 0 .. $#_ )
74     {
75         $_[$_] =~ s/^\s*|\s*$//g;
76         $_[$_] =~ s|</?b>||g;
77         $_[$_] =~ s|\s+| |g;
78     }
79 }
80
81 sub _field
82 {
83     my $data = $_[1];
84
85     HTML::Entities::decode_entities( $data );
86
87     my @matches = $data =~ m|<Q$_[2]\E>(.*?)</Q$_[2]\E>|sig;
88
89     wantarray ? @matches : $matches[0];
90 }
91
92 sub get_data
93 {
94     my $class    = shift;
95
96     require LWP::Simple;
97     my $data = LWP::Simple::get( $_[0] );
98     defined $data ? \ $data : $data;
99 }
100
101 sub get_url {
102     "http://syndication.sdmadiagroup.com/" .
103     "feeds/public/the_perl_journal.xml"
104 }

```

Listing 2

September 2004 PDF

Objective Perl: Objective-C-Style Syntax And Runtime for Perl	Kyle Dawkins
Scoping: Letting Perl Do the Work for You	David Oswald
Secure Your Code With Taint Checking	Andy Lester
Detaching Attachments	brian d foy
Unicode in Perl	Simon Cozens
PLUS Letter from the Editor	Perl News
	Source Code Appendix

August 2004 PDF

Regex Arcana	Jeff Pinyan
XML Subversion	Curtis Lee Fulton

OSCON 2004 Round-Up	Andy Lester
Molecular Biology in Perl	Simon Cozens
Pipelines and E-mail Addresses	brian d foy
PLUS Letter from the Editor Perl News	Source Code Appendix

Listing 3

```
#!/usr/bin/perl
package Local::Modulino::RSS::HTML;

use base qw( Local::Modulino::RSS );

require "RSS.pm";

__PACKAGE__->run() unless caller();

use HTML::Entities;

sub run
{
    my $class    = shift;

    foreach my $item ( $class->fetch_items )
    {
        my( $title, @articles ) = $class->get_issue( $item );

        print "\n<h3>$title</h3>\n\n<ul>\n";
        printf "<li><b>%s</b>", %s\n", @$_ foreach ( @articles );
        print "</ul>\n";
    }
}
```

Listing 4

```
<h3>September 2004 PDF</h3>

<ul>
<li><b>Objective Perl: Objective-C-Style Syntax And Runtime for Perl</b>, Kyle Dawkins
<li><b>Scoping: Letting Perl Do the Work for You</b>, David Oswald
<li><b>Secure Your Code With Taint Checking</b>, Andy Lester
<li><b>Detaching Attachments</b>, brian d foy
<li><b>Unicode in Perl</b>, Simon Cozens
<li><b>PLUS Letter from the Editor Perl News</b>, Source Code Appendix
</ul>

<h3>August 2004 PDF</h3>

<ul>
<li><b>Regex Arcana</b>, Jeff Pinyan
<li><b>XML Subversion</b>, Curtis Lee Fulton
<li><b>OSCON 2004 Round-Up</b>, Andy Lester
<li><b>Molecular Biology in Perl</b>, Simon Cozens
<li><b>Pipelines and E-mail Addresses</b>, brian d foy
<li><b>PLUS Letter from the Editor Perl News</b>, Source Code Appendix
</ul>
```

TPJ