



SpiderLogic
10000 Innovation Drive
Milwaukee, WI 53226

Phone: 414.290.8015
E-mail: info@spiderlogic.com

Where architects code and coders architect!



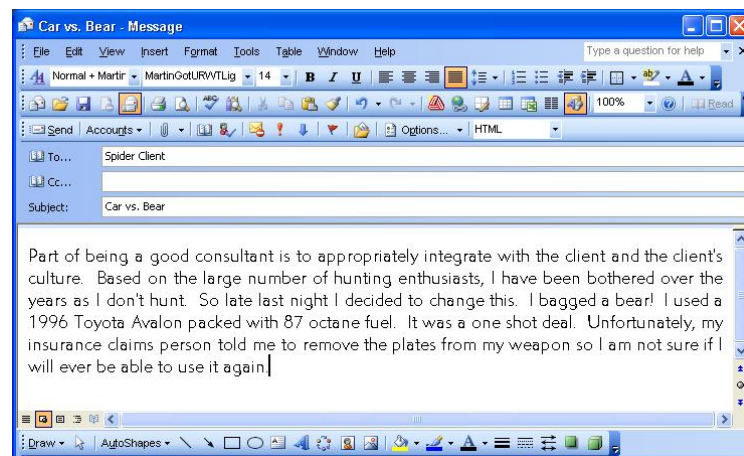
Car vs. Bear ... Bear Loses

By Quarterly Spin Staff

Sure ... it's common to hear about your neighbor or family member hitting the deer that came from nowhere and totaled their car. Steve Kronsoble, long time Spider Director, now can easily trump most deer encounter stories.

"My quest to get to 200,000 miles with my 1996 Toyota Avalon ended abruptly last night as I hit this poor bear head on at about 70 mph on my way home from northern Wisconsin. Hard to see a black bear on a black highway on a black night. Fortunately, I escaped unscathed. The bear – not so good." Kronsoble said.

Steve, who is an avid non-hunter, described his conquest in an e-mail to one of our clients located in northern Wisconsin.



Quarterly Spin



Volume 2, Issue 2
2009

Kids Names, Choose Carefully

By Dave Buettner

As part of my research for this newsletter, I came upon a blog written by fellow Spider Dan Tanner, in which he talked about the challenge of buying and reserving domain names for his kids.

This got me thinking about how the next generation may go through the name selection process. In our case, my wife and I followed the more traditional path of looking at names we liked, considering family names, thinking about unfortunate nicknames which could result from a name, and also making sure the names sounded OK together.

Certainly, there have been interesting names for kids used in the past. Frank Zappa's choices of

Moon Unit and Dweezil come to mind right away. My contention is that the next generation will need to get more creative as they choose names for their kids.

Right now, we are constantly being challenged by the User Name and Password requirements of various websites and software applications to come up with something unique. Sorry, user name "Dave" is not available. Or, please use a number or special character in your password.

Then, of course there is the added pressure of reserving your unique spot in cyberspace. Why wouldn't you want to make it easier for your son or daughter to have their own name as their domain name?

I predict that we will see new and different naming options. Sorry honey... David.com is already taken. Wait ... godaddy.com says that david5619.net is available. We can call him dave56 for short.

In addition, you may want to consider the addition of a special character or two in the middle name. This would make it easier to remember logins.

Example:
david5619 \$teven!



Inside this issue:

<i>Kids Names, Choose Carefully</i>	1
<i>Password Strength Validation with Regular String Expressions</i>	1
<i>Simple and Easy Site Monitoring with PowerShell</i>	2
<i>Car vs. Bear Bear Loses</i>	4



Password Strength Validation with Regular String Expressions

By Geoff Lane

Regular expressions are both complex and elegant at the same time. They can be made to look like someone was just randomly hammering on their keyboard. They are also an incredibly efficient and elegant solution to describing the structure of text and matching those structures. They are very handy for defining what a string should look like and as such are very good for use in data validation.

To validate a U.S. phone number, you might create a simple regular expression `\d{3}-\d{3}-\d{4}` which will match a phone number like 123-555-1212. Where regular expressions can become difficult though is when the format is not quite as clear cut. What if we need to support (123) 555-1212 and also 555-1212? Well, that's where things can get more complex. But this is not about vali-

phone numbers. In this post I will look at how to make assertions about the complexity of a string, which is very useful if you want to enforce complexity of a user created password.



Continued on page 3



Simple and Easy Site Monitoring with PowerShell

By Dan Tanner

In the *nix world, I really like monit for watching and managing small to medium apps. A while back we wrote an Intranet app in Grails, running on Tomcat on a Windows 2003 server. The client was hosting the app themselves, but didn't have any IT staff whatsoever, so monitoring and managing this app was a little tricky.

I searched around for something like monit for Windows, but didn't find any good free or cheap tools to do what it does, which is primarily to monitor and restart services upon failure. There's a million monitor and alert tools out there, but none that I found had a clean way to restart my Tomcat service if something went wrong. The closest thing I found was Hyperic's open source monitoring solution, but after 30 minutes of futzing around, it seemed too clunky to me for this small problem. I'm sure it's fine, and for bigger solutions it would probably fit well.

Then I ran across a couple PowerShell scripts that intrigued me - the first was for scraping a web page. After that find, it was just a matter of Googling to find the remaining piece - restarting a Windows service. Bingo! I hadn't done any PowerShell scripting yet, but it turns out to be a really sweet solution for this type of problem. All I had to do after that was to tweak the code to be a little more reusable, and I also created a simple Grails view that would exercise the integration features I wanted to include in my ping test. From there it was just a matter of calling the script on a regular basis. In my case, I used the Windows task scheduler. Here's what the script looks like:

```
$webClient = new-object System.Net.WebClient

#####
# BEGIN USER-EDITABLE VARIABLES

# the URL to ping
$HeartbeatUrl = "http://someplace.com/somepage/"

# the response string to look for that indicates things are working ok
$SuccessResponseString = "Some Text"

# the name of the windows service to restart (the service name, not the display name)
$ServiceName = "Tomcat6"

# the log file used for monitoring output
$LogFile = "c:\temp\heartbeat.log"

# used to indicate that the service has failed since the last time we checked.
$FailureLogFile = "c:\temp\failure.log"

# END USER-EDITABLE VARIABLES
#####

# create the log file if it doesn't already exist.
if (!(Test-Path $LogFile)) {
    New-Item $LogFile -type file
}

$startTime = get-date
$output = $webClient.DownloadString($HeartbeatUrl)
$endTime = get-date

if ($output -like "*" + $SuccessResponseString + "*") {
    # uncomment the below line if you want positive confirmation
    #"Success `t`" + $startTime.DateTime + "`t`" + ($endTime - $start-
    Time).TotalSeconds + " seconds" >> $LogFile

    # remove the FailureLog if it exists to indicate we're in good shape.
    if (Test-Path $FailureLogFile) {
        Remove-Item $FailureLogFile
    }
}

}
```

There's a lot more you can do with this script, like e-mail upon failure, or monitor multiple sites. Hopefully this helps some other folks out there with problems like this. It was a really nice experience for me; simple and powerful. So hats off to the authors of PowerShell - I'll be reaching for that tool more and more in the future.

Check out Dan's blog at Otherthanthink.blogspot.com



Password Strength Validations ... continued from page 1

The key to making password strength validation easy, using regular expressions, is to understand zero-width positive lookahead assertions (also know as zero-width positive lookaheads). Now that's a mouthful isn't it? Luckily the concept itself is a lot simpler than the name.

Zero-width positive lookahead assertions

Basically a zero-width positive lookahead assertion is simply an assertion about a match, existing or not. Rather than returning a match though, it merely returns true or false to say if that match exists. It is used as a qualification for another match.

The general form of this is:

```
(?= some_expression)
```

For example:

- The regular expression z matches the z in the string *zorched*.
- The regular expression z(?=o) also matches the z in the string *zorched*. It does not match the zo, but only the z.
- The regular expression z(?=o) does NOT match the z in the string *pizza* because the assertion of z followed by an o is not true.

Making Assertions About Password Complexity

Now that you know how to make assertions about the contents of a string without actually matching on that string, you can start deciding what you want to actually assert. Remember that on their own these lookaheads do not match anything, but they modify what is matched.

Assert a string is eight or more characters:
(?={8,})

Assert a string contains at least one lowercase letter (zero or more characters followed by a lowercase character):
(?=.*[a-z])

Assert a string contains at least one uppercase letter (zero or more characters followed by an uppercase character):
(?=.*[A-Z])

Assert a string contains at least one digit:
(?=.*[0-9])

Assert a string contains at least one special character:
(?=.*[\W])

Assert a string contains at least one special character or a digit:
(?=.*[\d\W])

These are of course just a few common examples, but there are many more that you could create as well.

Applying Assertions to Create a Complexity Validation

Knowing that these make assertions about elements in a string, but not about a match itself, you need to combine this with a matching regular expression to create you match validation.

- .* matches zero or more characters.
- ^ matches the beginning of a string.
- \$ matches the end of a string.

Put together ^.*\$ matches any single line (including an empty line). With what you know about zero-width positive lookahead assertions, now you can combine a "match everything" with assertions about that line to limit what is matched.

If I want to match a line with at least one lowercase character then I can use:

```
^(?=.*[a-z]).*$
```

The part that makes this all interesting is that you can combine any number of assertions about the string into one larger expression that will create your rules for complexity. So if you want to match a string of at least six characters long, with at least one lower case and at least one uppercase letter, you could use something like:

```
^(?=.*{6,})(?=.*[a-z])(?=.*[A-Z]).*$
```

And if you want to throw in some extra complexity and require at least one digit or one symbol, you could make a match like:

```
^(?=.*{6,})(?=.*[a-z])(?=.*[A-Z])(?=.*[\d\W]).*$
```

There you go. Now you can create regular expressions to check the complexity of passwords.

Check out Geoff's blog at Zorched.net