Windows/.NET

November 01, 2002

Regular Expressions in .NET

Email Print
Discuss Reprint

(Page 1 of 17)

This article will give an overview of .NET's regular expression class library, showing you how to use it to construct and execute regular expressions, as well as process the results. It will also give a short overview of a visual regular expression editor for .NET, RegexDesigner.NET.

You're about to release WonderApp.NET, the application that will change the way developers build applications, helping them to achieve dramatically faster development times with far less bugs. WonderApp.NET is full of cool functionality, including a feature to send feedback to WonderApp headquarters via e-mail, as shown in Figure 1.

Back at WonderApp headquarters, the mail server extracts message bodies from the incoming e-mails and forwards them on to FeedbackApp.NET, an application you've written to manage feedback, which includes bugs, feature requests, and general comments. One of your goals is to sift through these messages, looking for any positive feedback that might be suitable on the "Testimonials" page of WonderApp.NET's web site. Rather than manually trawling all feedback, you decide to automate the process by implementing a function in FeedbackApp.NET that finds sentences containing the string "cool". At a high level, the function scans the message body looking for any instance of the word "cool". The search for "cool" will take place on a sentence-by-sentence basis. When writing the function, you'll need to consider the following:

- What characters define the end of a sentence, and what happens if there aren't any?
- Is character case an issue?
- Depending on whether "cool" is at the front, middle, or end of the sentence, what characters are valid before and after it?

A programmatic solution to this problem requires two distinct code elements: text scanning and pattern matching. The text-scanning element accepts an input string, scanning through it one sentence at a time. The pattern-matching element focuses the scanning to specifically search for "cool" and ensure that it's in a valid sentence.

The resulting function would tightly integrate both elements into a single piece of code. Although it could be refactored to scan for sentences containing any sequence of characters, it would take a great deal more effort to extend the solution to handle broader parsing problems, including:

- Validating a string that is in a particular date/time format, such as dd/mm/yyyy.
- Finding **<meta>** tags within an HTML document.
- Counting the number of occurrences of a "word" in a given string.

Regular Expressions

Fortunately, regular expressions are designed to solve exactly these kinds of problems by providing a rich notational language for defining the text patterns that an optimized text-processing engine looks for. The design readily lends itself to solving a broad variety of text-parsing problems, and reduces developer effort to just defining what the pattern should be; the text-processing engine does the rest. Any pattern, or regular expression, can include literal characters, metacharacters, or any combination of the two.

Literal Characters

If the regular expression (regex) engine finds an exact copy of the literal characters in the input string, they are considered a match. For instance, if you wanted to look for the word "cool" in a bunch of text, you could create a regex comprised entirely of the following literal characters:

```
cool
```

Behind the scenes, the regex engine scans the input string from left-to-right, looking for a sequence of characters that match. Figure 2 shows the results, highlighting the matched characters.

Alternation

Unfortunately, due to the case-sensitive nature of regexes, "cool" won't match instances of "Cool", which could be found at the start of a sentence. To match either "Cool" or "cool", you could use the regex alternation metacharacter, '|':

```
Cool|cool
```

Metacharacters are the notation or language that gives regex its power, covering types, ranges, and numbers of characters to match. '|' is used to separate two or more subexpressions, any of which are allowed to match at that position in the input string, such as the previous example that will match either "Cool" or "cool". With group metacharacters, '(' and ')', you could rearrange the regex and achieve the same result:

```
(C|c)ool
```

Grouping

This technique uses grouping metacharacters to contain the alternatives, both lower and upper case "c", either of which must then be followed by "ool" before being considered a successful match. The regex's meaning would be different if they weren't included. Given:

```
C|cool
```

either "C" or "cool" will be matched.

Options

Another approach to handling case insensitivity is to use the **IgnoreCase** regex option. Options alter the overall regex matching behavior and, you guessed it, **IgnoreCase** turns off regex's default case sensitivity. The **IgnoreCase** option can be specified by prepending the short-form of the option '**i**' in a special construct at the beginning of the regex, like so:

```
(?i)cool
```

This inline syntax applies the option to an entire regex. However, an alternative syntax allows you to apply an option to either the whole expression or a subexpression:

```
(?i:c)ool
```

One advantage derived from using options is the improvement of readability. Compare the previous regex to a regex that anticipates any possible combination of lower- and upper-case characters:

```
(C|c)(O|o)(O|o)(L|l)
```

The latter ends up being both harder to type and to read. .NET regex offers several options, all of which are listed in Table 1.

Character Classes

Now that you can find "cool", the next step is to find a sentence containing it. Let's start by deciding on the characters that are valid at the end of a sentence, including '.', '?', or '!'. You could use alternation:

```
(\.|\?|!)
```

This expression works as expected, but why are there backslashes? Regexes contain both literal characters and metacharacters. One potential match you'll run into is where you need to literally match a character that also happens to be a metacharacter, such as '.' and '?'. Escaping metacharacters with '\' forces them to be treated literally. One potential side effect is a reduction in readability: The use of '\' makes it harder to work out what the subexpression does. Also, imagine if you needed to match a larger number of characters than those included in this example. Your regex could get quite ugly very quickly.

Fortunately, regex offers a nicer way to group sets of characters together: the character class. A character class is considered to be the set of characters included within a pair of left and right square brackets, relieving you from using escape characters (unless you need to match '[' or ']'). When rewritten as a character class, the end-of-sentence alternation example would look like:

```
[.?!]
```

At this position in the input string, the character class tells regex to check for one occurrence of any of the characters it contains. The subexpression matches four times against the input string, as is shown in Figure 3.

When you do need to handle a large range of characters, character classes offer a very simple way to describe them, using '-'. The alphanumeric range of characters, both upper and lower case, is defined by this regex:

```
[a-zA-Z0-9]
```

A character class can also be negated. If you want to match all nonalphanumeric classes, you take the previous regex and insert '^' in front of the other characters inside the square

brackets:

```
[^a-zA-Z0-9]
```

Table 2 lists the common character classes.

Quantifiers

Character classes are compared against one character in the input string at a time. Regex quantifier metacharacters, however, allow you to compare against one or more consecutive characters. Quantifiers must directly follow the subexpression they relate to. The following example uses the '+' quantifier to specify one or more consecutive matches on the end-of-sentence characters:

```
[.?!]+
```

The multiple contiguous occurrences of '!' are treated as a single match, resulting in the total number of matches found being reduced to three, as shown in Figure 4.

'?' and '*' are the quantifiers for zero or one and zero or more matches, respectively. Be careful with either of these metacharacters: If their associated subexpressions are not found in the input string, the match is still considered successful because it found zero instances. Table 3 contains the complete set of quantifiers in .NET.

You've now seen the basic range of metacharacters needed to create a regex capable of finding one or more sentences with the word "cool", which looks like:

```
(?i)[^.?!]*cool.*[.?!]
```

This expression is looking for:

- Any sequence of characters that aren't sentence terminators, followed by
- "cool", followed by
- any character that isn't a new line character, followed by
- an end-of-sentence terminator,
- all indifferent to case.

The new regex finds only three matches this time, shown in Figure 5.

Single Line

You'll notice that both carriage-return and line-feed characters are unmatched at the beginning of a sentence, since they are not end-of-sentence characters. But given that e-mail is free-text, there could be any number of whitespace characters in a valid sentence. If the first sentence were split over two lines, the regex wouldn't find the desired number of matches, demonstrated in Figure 6.

Now there are only two matches. The first sentence doesn't match because it's prevented by the '.' metacharacter, since '.' in a regex won't match '**\n**'. Such special treatment of the new line character demonstrates regex's line-oriented behavior. One approach to matching the first sentence is to use alternation to allow a check for either '**\n**' or '.', the net result effectively ignoring new lines and treating the input string as a single line. Alternatively, you could override the default regex behavior for '.' with the **SingleLine** option, which allows '.' to match '**\n**' as well:

```
(?si)[^.?!]*cool.*[.?!]
```

But solving this problem introduces another. The result now only returns one match, as shown in Figure 7.

Greedy vs. Nongreedy

Why didn't the regex stop matching when it found the first occurrence of an end-of-sentence character? Both '*' and '+' are known as "greedy" quantifiers, since regex will try to match as much as possible when it encounters either. In this example, after the first instance of "cool" is matched, the '.*' matches all the way up to the character before '!', which is matched by '[.?!]', illustrated in Figure 8.

Our goal, though, is to match sentence-by-sentence and not the entire input string in one fell swoop, potentially including sentences that don't contain "cool" between sentences that do. Fortunately, you can specify "nongreedy" matching by placing the '?' metacharacter after any quantifier, including the '**?**' quantifier. With groups added for readability, the new "restrained" regex:

```
(?si)([^.?!]*?)(cool)(.*?)([.?!])
```

successfully matches the input string three times, shown in Figure 9.

System.Text.RegularExpressions

The regular expression is complete. All we need now is a way to execute it against a feedback message body. The .NET Framework includes a rich regular expression, under the **System.Text.RegularExpressions** namespace. The **Regex** class is located here, and provides all the functionality you need to create and execute regexes, and process the results. You could use **Regex.IsMatch()** to determine whether a regex finds a match in an input string:

```
using System;
using System.Text.RegularExpressions;
...
string input =
    "This product\r\nis so cool! I love
    how cool it is!\r\nCheers, DevGuy";
string pattern =
    "(?si)([^.?!]*?)(cool)(.*?)([.?!])";
bool isMatch =
    Regex.IsMatch(input, pattern);
...
```

Regex.IsMatch() is a static method that accepts both regex pattern and input string parameters, returning True if at least one match was found, False otherwise. .NET offers a programmatic way to specify regex options, such as **IgnoreCase** and **SingleLine**, like so:

```
...
string input =
    "This product\r\nis so cool! I love
    how cool it is!\r\nCheers, DevGuy";
string pattern =
    "([^.?!]*?)(cool)(.*?)([.?!])";
RegexOptions options =
    RegexOptions.IgnoreCase |
    RegexOptions.Singleline;
bool isMatch =  Regex.IsMatch(input,
    pattern, options);
...
```

Regex.Match()

If you wanted to check that an input string was a valid date, a simple true/false value would be adequate. However, to get the "cool" sentences, you'll need access to the substring that was matched. **Regex.Match()** returns a **Match** object that provides detailed information about a match, including its (text) **Value**, **Index**, and **Length**,

relative to the start of the input string:

```
...
string input =
    "This product\r\nis so cool! I love
     how cool it is!\r\nCheers, DevGuy";
string pattern =
    "(?si)([^.?!]*?)(cool)(.*?)([.?!])";
Match match =
    Regex.Match(input, pattern);
...
Console.WriteLine(string.Format(
    "Matched?: {0}", match.Success));
Console.WriteLine(string.Format(
    "Value : {0}", match.Value));
Console.WriteLine(string.Format(
    "Index : {0}", match.Index));
Console.WriteLine(string.Format(
    "Length : {0}", match.Length));
...
```

A **Match** object is returned whether or not the match was successful. If a match is successful, the returned **Match** object represents the first occurrence of the match in the input string, or empty values otherwise; i.e., **Value = ""** and **Index** and **Length = 0. Match.Success** offers a shortcut to using these values to determine whether the match was successful or not, returning True if so and False otherwise. A side effect of using either **Regex.IsMatch()** or **Match.Success()** is that they conceal the exact number of matches found because they only report whether there was at least one match.

Match.NextMatch()

However, you can use **Match.NextMatch()** to return the next match in the input string, starting from the end of the current match. If you place **NextMatch()** inside a while loop, you iterate through the entire set of matches, checking **Success** at each iteration until false. The following example demonstrates this, iterating each match on a "cool" sentence:

```
...
string input = "This product\r\nis so cool!
   I love how cool it is!\r\nCheers, DevGuy";
string pattern =
    "(?si)([^.?!]*?)(cool)(.*?)([.?!])";
Regex re = new Regex(pattern);
Match match = re.Match(input);
...
int matchCount = 0;
while( match.Success ) {
  matchCount++;
  ...
  match = match.NextMatch();
}
Console.WriteLine(string.Format(
    "Total Matches: {0}", matchCount));
...
```

This technique is analogous to read-only/ forward-only firehose data access, like using ADO.NET's DataReader. As with the database cursor, you can only move forward through the "match set," one match at a time. While regex allows you to reverse the match order with the **RightToLeft** option, there's no way to apply it after a **Regex** object has been constructed. Also, **Match.Success/Match.NextMatch()** does not provide a mechanism to index any match or any metadata describing the match set, including a match count, which the previous code had to implement manually.

Regex.Matches()

If you need metadata, ad-hoc indexing, and the ability to iterate the match set in any direction, use **Regex.Matches()**, which returns a **MatchCollection** that offers all of these features, as well as **for-each** enumeration. The following sample is the collection equivalent of the **Match.Success**/**Match.NextMatch()** approach:

```
...
string input = "This product\r\nis so cool!
    I love how cool it is!\r\nCheers, DevGuy";
string pattern =
    "(?si)([^.?!]*?)(cool)(.*?)([.?!])";
Regex re = new Regex(pattern);
MatchCollection matches = re.Matches(input);
...
foreach( Match match in matches ) ...
Console.WriteLine(string.Format(
    "Match # : {0}", matches.Count));
...
```

Extracting from Groups

With the feedback parser completed, you released the product and, after several months, WonderApp.NET collected some great testimonials. It also collected enough feedback to form a list of compelling features for WonderApp.NET v2.0. Due to the success of Version 1.0, you redevelop the application and release it for beta testing. Like feedback, bugs can be submitted via e-mail sent to WonderApp headquarters where the e-mail body is extracted and, this time, sent to your bug tracking application, BugTrackerApp.NET. The bug report e-mail contains both "Description:" and "Repro:" placeholders, designed to help the users frame their bug report, as shown in Figure 10.

BugTrackerApp.NET will need to extract both Description and Repro text from the e-mail body before being stored. You're interested in capturing whatever the user enters but, given that e-mail is free form, there could be any amount of whitespace between the important parts, being split over multiple lines and in mixed character case. Regular expressions are sounding like a great candidate, and you start by devising the following regex to match a bug report e-mail body:

```
(?si)\s*(Description\s*:(.*))(Repro\s*:(.*))
```

The problem with this regex is that it will match against an entire bug report message body while our goal is to individually extract both the description and repro substrings. Luckily, regexes and .NET provide the mechanisms you need to do so and, even better, you only have to add some groups to the regex to do it. You've already used groups for alternation and readability, but groups can also be pulled out of a match for further processing by your application. This is possible because regex remembers the value that each group's subexpression matched. Regex assigns each group a number, starting from one and counting up in left-to-right order of opening parentheses. The bug report regex contains the following numbered groups; see Figure 11.

Group and GroupCollection

When you call **Regex.Match()**, .NET transcribes a regex's groups into a single **GroupCollection**, which you can access. The direct correlation between group number and **GroupCollection** index allows you to select the appropriate group values. Each **Group** object in the collection exposes a **Value** property that contains the text matched by the group's subexpression. **GroupCollection** always contains Group 0 and then one group object for each group in the regex. Group 0 has a value equal to its parent **Match**'s value. The other groups are added in order of group number, starting at index 1, enumerated as shown here:

```
...
string input = "Description:The app is
    broken.\r\nRepro:Double-click the app icon.";
string pattern =
```

```
    @"(?si)\s*(Description\s*:(.*))(Repro\s*:(.*))";
Match match = Regex.Match(input, pattern);
...
int groupCount = 0;
foreach( Group group in match.Groups ) {
  Console.WriteLine(string.Format(
    "Group# : {0}", groupCount));
  ...
  groupCount++;
}
Console.ReadLine();
...
```

This code enumerates the **GroupCollection**, displaying information for each group found. The direct correlation between group number and **GroupCollection** index allows you to select the appropriate group values. You can use the group number/index to extract the text in Groups 2 and 4 of the bug report regex:

```
...
string input = "Description:The app is
    broken.\r\nRepro:Double-click the app icon.";
string pattern =
    @"(?si)\s*(Description\s*:(.*))(Repro\s*:(.*))";
Match match = Regex.Match(input, pattern);
...
Console.WriteLine(
    "Description: " + match.Groups[2].Value);
Console.WriteLine(
    "Repro : " + match.Groups[4].Value);
...
```

While this code is retrieving the appropriate data, it depends on hard coding group numbers, ultimately tying your code to the group's physical location. Such code is brittle when considering that a regex's groups could be moved, added, or removed, thereby changing the indices your code refers to. But what if another developer were to rearrange the groups in the bug report regex:

```
(?si)\s*Description\s*:(.*)Repro\s*:(.*)
```

Now your extraction code would return the wrong values, since it's referencing the wrong groups: Group 2 is now Group 1 and Group 4 has been removed. **GroupCollection** doesn't help either, since it returns an empty group when it's referenced by an out of range index. This is confusing because even though Group 4 doesn't exist, it will return a value, albeit empty. Hiding indexing errors makes it tricky to find this type of bug, since **GroupCollection** shields it from you.

Named Groups

Fortunately, .NET offers named groups that provide a layer of indirection between a group and its physical regex location. The named group version looks like:

```
(?si)\s*Description\s*:(?<descr>.*)Repro\s*:(?<repro>.*)
```

Now, if any groups are altered, the code will still refer to the correct groups.

```
...
string input = "Description:The app is
    broken.\r\nRepro:Double-click the app icon.";
string pattern =
    @"(?si)\s*(Description\s*:
    (?<descr>.*?))(Repro\s*:(?<repro>.*))";
Match match = Regex.Match(input, pattern);
```

```
...
Console.WriteLine("Description: " +
    match.Groups["descr"].Value);
Console.WriteLine("Repro : " +
    match.Groups["repro"].Value);
...
```

If a regex has a combination of named and numbered groups, the numbered groups are counted by regex before the named groups, shown in Figure 12.

Replacement Strings

So far, you've seen how to retrieve matches and access numbered and named groups for more granular match result values. But, matches also form the basis of a variety of other regex operations, including replacement and splitting. Replacement will find every match in an input string and replace it with another one. This might have come in handy during the feedback phase when you received a few "lame" comments — a lot more than would look good at your next review. To take the sting out of it, you could have used **Regex.Replace ()** to replace every instance of the word "lame" with "OK", shown in the following sample.

```
...
string input = "Your application is lame!";
string pattern = "lame";
string replace = "OK";
...
Console.WriteLine("Original version: " + input);
Console.WriteLine("Nice version: " +
    Regex.Replace(input, pattern, replace));
...
```

This sample outputs the following results:

```
Original version: Your application is lame!
Nice version: Your application is OK!
```

Replacement Patterns

Replacement strings replace matches with other strings. You can also do the reverse by replacing or substituting templates in a destination string with values from a match. As an example, when you find suitable testimonials, it might be nice to return a user's e-mail address with the matched sentence in a nice format, such as:

```
[e-mailaddress] said: "<matched sentence>"
```

Again, look no further than .NET regexes and replacement patterns. A replacement pattern uses both literal and special substitution characters to construct a new result string based on a combination of the two, where values from the source match are merged into the result via the substitution characters. There are several substitution characters, including named and numbered group substitutions that suck out the last value the named or numbered group captured. Substitution occurs after the match has taken place, since it needs to know what text the groups captured. Subsequently, you can substitute Group 0, which is what we'll do to build the formatted feedback, rather than adding a new group to cover the entire regex. Given the "cool" sentence finder from earlier:

```
(?si)(.*?)(cool)(.*?)([.?!])
```

and the replacement pattern:

```
joeuser@company.com said: "${0}"
```

a replacement on the sentence "This product is so cool!" returns this string:

```
joeuser@company.com said: "This product is so cool!"
```

Using named groups instead of numbered, you can achieve the same result more robustly as shown with the following code:

```
...
string input = "This product is so cool!";
string pattern =
    "(?si)(?<quote>(.*?)(cool)(.*?)([.?!]))";
string replace =
     @"joeuser@company.com said: ""${quote}""";
...
Console.WriteLine("Original match: "
    + Regex.Match(input, pattern));
Console.WriteLine("Replaced : "
    + Regex.Replace(input, pattern, replace));
...
```

Split

Split is another interesting regex feature that, like replace, relies on matching to do its work. Split will divide an input string around the matched substrings and return the results in a string array. The following code sample demonstrates how you could have used split functionality to extract the "Description:" and "Repro:" data from bug reports, instead of groups:

```
...
string input = "Description:The app is
    broken.\r\nRepro:Double-click the app icon.";
string pattern =
    "(?si)description:|repro:";
string[] splits =
    Regex.Split(input, pattern);
...
Console.WriteLine(
    "Description: " + splits[1]);
Console.WriteLine("Repro : " + splits[2]);
...
```

RegexDesigner.NET

Regular expressions are a powerful way to solve a lot of text-processing problems. This power is also what makes them sometimes quite difficult to use, let alone remember. Towards that end, we built a little tool to visually build and test regular expressions. It's called RegexDesigner.NET (RegexD for short) and it's shown in Figure 13. RegexD can be downloaded for free at www.sellsbrothers.com/products/#regexd. The best way to develop your understanding of regexes is by doing them. Using RegexD allows you to experiment before writing your regex code (in fact, RegexD will write the code for you).

Summary

This article has given you an overview of regular expressions and how they are implemented in .NET. Regexes are a rich text-processing technology that can solve a broad variety of find, replace, and split problems. .NET exposes a set of classes under the **System.Text.RegularExpressions** namespace that allow you to construct and execute regexes, and browse match and group results. If you don't want to roll your own custom pattern-matching language and text-parsing engine, you will find a great alternative in regular expressions. Source code examples

demonstrating the functions of the regex library are available online.

References

- RegexDesigner.NET, http://www.sellsbrothers.com/products/#regexd.
- Microsoft Developer Network (MSDN) Online, http://msdn.microsoft.com/.
- Online Regular Expression Library for .NET, http://www.regxlib.com/.
- Friedl, Jeffrey E.F. *Mastering Regular Expressions*, O'Reilly, 2002.

Michael Weinhardt is a software engineer at SERF, a retirement fund company where he develops .NET Windows and web-based applications. He can be reached at mikedub@optusnet.com.au. Chris Sells is an independent consultant specializing in distributed applications in .NET and COM, as well as an instructor for DevelopMentor. He's written several books including ATL Internals*, which is currently being updated for ATL7. He's also working on* Essential Windows Forms *for Addison-Wesley and* Mastering Visual Studio .NET *for O'Reilly.*

1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 Next Page

RELATED ARTICLES

Whitebox Security Testing Using Code Scanning
Deploying SQL Server Everywhere
Loose Ends
Understanding Dependency Injection
The TVisto Media Center

TOP 5 ARTICLES

Quick-Kill Project Management
The Essential Unified Process
Survey Says: Agile Works in Practice
It's (Not) All Been Done
Lock-free Interprocess Communication

enterprises and government agencies.