2018 STC Technical Communication Summit
20–23 May 2018 • Orlando, Florida

Using Regular Expressions with Madcap Flare: Putting Your Searches on Steroids

TOC

Robert Delwood

Regular expressions take find and replace to new levels. You can find text patterns, not just exact text. With patterns you can find, replace and even format all phone numbers, replace all span tags in a Flare document ("<span_1>my text" with "my text"), find every e-mail address in a document all documents names, all dates formats, and words you're not even sure how they're spelled, like Jeffrey, or Jeff, Jeffery, and Geoffrey. Do each of these in a single search.

This ability, using a notation called regex, is described as find on steroids. Learn how Madcap Flare supports find using regular expressions. It's not limited to just Flare. This works with any text based product. With other tools, such as NotePad++, you can make changes to one or more files at the same time. It's versatile enough that writers and programmers alike can use it, and opens a new world to find and replace.

Everyone is familiar with the Find feature. It's a ubiquitous feature that almost every application has built in, and that we've come to rely on frequently. Yet, for as simple and powerful the feature is, it seems few take full advantage of it. For instance, think about the times in the last month you've done a find within a document. Chances are it's a common event. Now, think about the times in the last two months you've done a find on two or more documents at the same time, a multiple document find. This number is likely dramatically less than the first. And finally, in the last three months the number of times you've done a find and replace on two or more documents at the same time, a multiple document replace. Likely, that's almost zero. So why the precipitous fall between each case? I'd like to talk about that. This paper is about reviewing the lowly find feature and elevating it to a new, high status.

I've always maintained that writers are about being a put upon group. We will never get the number of tools we really need. Tools here are defined as small applications or features within an application that help writers do their job. Those are the ones for the low level tasks, such as formatting text, listing files in a directory, and especially automating tasks. These tasks may be onetime events, unique to each writing group, even idiomatic for individual writers. It's understandable then why we will never get those tools. They're too minor or uncommon for companies to include them. Except they may not be minor or uncommon for the writer. All much more the reason to fully utilize the ones we have. And regular expressions should be added to our list.

What Are Regular Expressions?

Regular expressions, also called regex, are a pattern-finding notation. It's a find feature but instead of finding matches based on exact text, it can find patterns. For example, we're all used to conventional finds when you enter the exact text, such as *ubiquitous* or *idiomatic*, and the find locates instances of that text exactly in the document. There may be slight options variations such as upper and lower case (*Ubiquitous*/*ubiquitous*), or whole word only, but those don't change the point here. It's still looking for a letter to letter match.

It's pattern finding. Regex looks for patterns or specific groupings based on wild cards. As an example, sometimes you want to find something but you don't know what you're looking for. Suppose the document is full of serial numbers, defined as six numeric characters in a single string, such as 456923. You don't know what the serial number is ahead of time, and perhaps you don't even care. You just want to find them. That's where wild cards come in. A wild card is where a single character

matches more than one character. In a conventional search, one character matches one character. For *ubiquitous*, u matches u, b matches b, and so on. With a wild card, one character matches multiple characters. The regex notation **\d** matches any numeric digit, the characters 1 through 9 and 0. So instead of having to specify the exact number, such as a conventional find would do, you could enter **\d\d\d\d\d\d**. The series of six numeric wild cards would match any six numeric characters.

It's a notation. Regex is considered a notation and not a computer language. It doesn't contain any built in logic, any predefined functions, and can't use language features such as If statements or looping. It uses keyboard characters to define the matching sequence. That means it also shares some of the letters and symbols in common. In the numeric wild card case of **\d**, it uses the backslash to differentiate this meaning from the conventional meaning of the letter d.

However, regex does need to be run by something. It may be built into the application. Madcap Flare supports regex, as does the application NotePad++, an enhanced version of Windows' venerable NotePad. Word supports a wild card notation although it's not generally considered as regex. All three support both find and replace wild card features. Other applications support just regex features, and may be needed for specialized or specific find issue.

Regex is more of a guideline that an actual product or even a standard. Since applications have to support and run regex, there is some variation from the regex ideal. That means each application's version is slightly different and will use different names. Grep, PCRE (Perl Compatible Regular Expressions), and almost each language such as Microsoft .NET, UNIX, Java, and Python will have their own variations. The products are largely similar; perhaps 80% are the same in each version. The differences will reflect the language's concentration on different material such a string or mathematical processing.

Even though regex isn't a language, it can be used by languages. Almost every language supports regex and can be used to make regex calls. In fact, the true power is regex is best seen in a language where repetition and processing can automate the process. Imagine having a scientific instrument that can collect thousands of inputs per second. An application can loop through that data, perhaps into the billions, and find and catalog highly complex patterns. In a more practical example, imagine being able to go through a long document, finding every Web site reference, cataloging it, and reporting if the site is available or not. Regex is able to find the pattern match, and the language provides the repetition, logic, and automated support.

How to Use Regular Expression

Regex is a rich, versatile, and sometimes complex notation, so it's not possible to cover every concept in this brief paper. Instead, I want to present fundamentals so that you can understand what's possible, and then it's just a matter of matching up terms to meet your needs.

There are five important concepts: Literals, wildcards, range, repetition, and position.

Literals. Literals are the keyboard characters. These are the letters, numbers, and symbols that we're used to with conventional searches. You can use only literals for your regex searches. You'd lose the power of regex. This isn't limited to the 96 characters on a keyboard, but also includes the possible 1,114,112 Unicode characters.

Wildcards. Wildcards are single characters that can match multiple characters. This is mentioned earlier with the **\d** wildcard matching any numeric digit. Regex has a rich set of wildcards.

.	Any character
\s	Any whitespace
\S	Any non-whitespace
\d	Any digit
\D	Any non-digit
\w	Any word character
\W	Any non-word character

The period, for example, is the most versatile wildcard because it matches any single character. For example, **c.t** would match the pattern c, any character, and t. This includes the obvious ones of *cat*, *cut*, and the time abbreviation *cst*, but it also can find matches anywhere within a word such as *citizens*, *Scottish*, and *classification*.

It's useful to point out that any regex notation can be made into a literal by using what is called an escape character. This is a backslash (\) immediately before the character. If you wanted to find the period at the end of a sentence, you couldn't really use the period notation, since that's a wild card and would not only find a literal period but also any other character. In this case you'd use **\.** to indicate

finding a literal period. The same is true for finding a backslash character, such as ****.

However, often you're not interested in finding just any character, you want to find a specific character or a kind of character. We've already seen the digit wild card of **\d**. The **\D** (with an uppercase D) finds any character other than a digit. **\w** finds any word character. That is any letter (which includes keyboard and Unicode) number, and underscore. **\W** is any character that is not considered a word character. **\s** is any white space character. This is any space character and also tabs, line feed, carriage return, and new line characters. **\S** is any non-white space character or any character not included in the white space category.

Range. Hopefully the power of the regex is started to be seen. Combining literals and wildcards adds a versatility that a conventional find can't match. For example, finding any serial number is an improvement over the conventional find. And you can combine literals with wildcards to find more specific serial numbers, such as using **1/d/d/d/d/d** to find all serial numbers beginning with the number 1. Even so, just those two are still limited. With the range notation, you can set a range or a group of characters to find. This notation finds any character inside hard brackets, (**[]**). This can be any set of characters, explicitly listed like **[AEIOU]**. It can be a range, with the first and last characters noted like **[a-z]** or **[1-0]** (zero being literally larger than 9). It can be both, such as **[1-0ABC]**, which matches any number, or the upper case A, B, or C. You can use wild cards, so that **[\dABC]** matches the same characters as **[1-0ABC]**. The match is successful if any character occurrences within the hard bracket set.

For example, **[24680]/d/d/d/d/d** finds any serial number beginning with 2, 4, 6, 8, or 0. **[a-z]** finds any lower case letter. **[a-zA-Z]** finds any letter, lower or upper case. And they can be mixed, too **[a-pr-z]** finds any lower case letter except q. Notice the ranges are actually a-p, and again for r-z.

Repetition. This is the ability to repeat a sequence. To find six digit serial numbers, we've been using the **\d\d\d\d\d\d** notation sequence. An obvious problem with that is there are six repeated values. Sequences like that should almost always be avoided. They're hard to maintain, duplicate, and change. Regex uses the curly brackets (**{}**) to indicate the number of times the immediately preceding notation should repeated. The serial number sequence can now be shortened to **\d{6}**.

Regex provides two ways to provide repetition: explicit and implicit.

Explicit repetition notation uses the curly brackets. If one number specified, then the pattern will be repeated exactly that number of times. So **\d{6}** as mention matches exactly six digits. This means a digit string of five or less will not be matched at all, and if the digit string is more than six, then only six will be matched. That would be the first six digits unless otherwise specified. If two numbers are inside the curly brackets, then the first number is the minimum number needed to make a match, and the second one is the maximum number that will be matched. So **\d{2,6}** matches two to six digits in a row. If you're looking for a user name, defined as six to 12 alphanumeric characters, you could use **[a-zA-Z0-9]{6,12}**. This would fully match *PvtDancer98*, match the first 12 characters of *PrivateDancer98*, and not match *PvtDc* at all.

Implicit repetition notation finds sequences based on zero, one, or more occurrences. The notation immediately preceding the following characters will attempt to be found.

?	Finds zero or one occurrences
*	Finds zero or more occurrences
+	Finds one or more occurrences

For example, **<li class** would find all occurrences of that but only if there was exactly a single one space between the two elements. **<li +class** is more versatile because it finds the same two elements but now it does so even if there are two or more spaces between them. **Me** will find the two letters adjacent to each other as in *me*, *meliorate*, and *presentiment*. **M.e** finds *m* and *e* but with any single character in between them such as *Ramses*, *climbed*, and *squirmier*. Changing it to **m.*e**, with the implicit asterisk notation now finds *m* and *e* but with any number of characters in between. This includes *basement*, *lachrymose*, and *militantness*. The **?** notation (which has two uses and the second one is mentioned later) makes the proceeding character optional. As examples, **ou?r** would find the English and American spellings of the words *colour*/*color*, and *humour*/*humor*.

Position. So far, most of the searches above will match the sequence anywhere in the word, whether it's the entire word itself or a limited sequence within a word. For example, **\d{6}** matches 123456. It also not only matches the first six digits in the seven digit 1234567 (123456) but also the last six digits in the same word (234567). This is a case conventional

finds don't often have to deal with. Most of those have an option to find whole word matches, perhaps on by default. Regex has two important position finding characters:

^ Start of word

$ End of word.

The caret (**^**) will find matches that start at the beginning of the word. This means **^\d{6}** will only find a match if it's at the beginning of the word, the 123456 from 1234567 and not the second instance, 234567. The dollar sign (**$**) is the opposite and matches only at the end of the word. **\d{6}$** matches the sequence at the end of the word, 234567 from 1234567, and not 123456. Using both will find the match only if it's at the beginning and the end. That is, if it's the entire word. **^\d{6}$** matches only the word 123456, and will not match anything from 1234567.

When to Use Regex

The building blocks are all in place now. Don't worry about not understanding them yet – that comes with time. If all this seems excessive for just text, perhaps it is. Most of the time when looking for text, you'll know what you're looking for. The serial number example may be more of an exception. That could be a reason why regex isn't not more popular among writers. But content or body copy isn't the only kind of text writers use. Regex's real power can be applied to HTML and XML. Madcap Flare users write mostly in the XML editor, or the WYSIWYG editor. Many don't think in terms of HTML or XML, and use those only in few cases. Regardless, we make an assumption that that is well formed and consistent, but this is far from the truth. For example, anyone who's ever worked with Flare's behind the scene code understands how complicated and inconsistent it gets. Given the possible formations, the machine generated code, and especially if Word was ever involved, it seems impossible to find anything. It's not uncommon to see Flare's HTML like:

** Who steals my pursesteals trash; 'tis something, nothing; **

Example 1: Sample HTML source code for the following procedures.

Clearly, this is messy code. Beyond some of our compulsion to have clean, organized code, this formatting may affect how the text displays, depending how **span_1** is styled. Assuming that's the case, it needs to be cleaned up. There are two cases that this is helpful in.

The first case is that the **span_1** and **span_2** is unwanted and should be ****. It may have been introduced during a file import process. In that case, we have the additional concern that there might be a **span_3**, **span_4**, and so on. Or it may have been intentional at one point but now you want it changed. In any case, we want to change all those spans to **span_lago_reference**. A simple find and replace might work if we were sure there were only a **span_1** and **span_2**. We want to catch all the instances include unexpected ones. Regex simplifies this. You can find **<span = class="span_\d{1,}">** and replace with ****. The find matches the literal part of <span class="span_ with any number of digits after that. It nicely covers unexpected instances like span_27 or span_999.

The second case is that you don't want those numbered span tags at all. The problem now is that the span tags are actually pairs of tags, with the opening tag, text, and a closing tag. It may be easy enough to find and remove the opening span tag, I've already pointed out how to find those, but the search doesn't find the closing tag. In addition, there's nothing unique about the closing tags. That means you can't differentiate one closing tag from another and you can't just remove all **** tags because there are some we want. For example, we want to keep ****. Again, regex can do that. It takes two expressions. One for the find and another for the replace.

For the find segment, we're going to craft an expression that includes both tags at the same time. That means it also includes the text between them, something we want to keep anyway. The find would be **(.*?)<\/span>**. This expression can be broken up into three components of **, (.*?), and <\/span>**. We've seen the first part of **** in the previous example. It finds text like span_1 or span_300. The last part catches the closing span. The interesting part is the middle expression of **(.*?)**. The period was discussed as being a wild card that matches any character. The asterisk is the repetition marker for zero or more occurrences. So together **.*** finds zero or more occurrences of any character. You can see

how that the expressions matches everything in between the two tags.

The problem is how to specify which closing tag to end with. In our example, there are four closing span tags. At worst, the expression could find all the characters from the first opening tag to the last closing tag. We're interested in the characters from the first opening tag through only the first closing tag. For that, regex introduces the concept of lazy and greedy searches. By default, all searches are greedy, which means it will try to get the greatest number of characters that match our pattern. In contrast, a lazy search attempts to find the match with the fewest number of characters. That's the one we need. The question mark following a repetition notation indicates to make the search lazy. And, yes, they double up on meanings for characters The result here is that our search matches the first closing span tag after the opening span tag. That would be **Who **. It's hard to notice here but it includes a space after the word *Who*.

For the replace segment, we're going to selectively add back the body text we found. In the first case example, a replacement was made using fixed text, (****). Most of us are accustomed to using fix text with replaces. But here, it's going to be based on the actual found text. That is marked in the find expression by the parentheses around the term, such as **(.*?)**. These parentheses form what's called a replacement group. If there's only one set of parentheses, that's the first replacement group, and noted in the find text as **\1**. If there's a second set of parentheses that would be noted as **\2**, and so. So for the find text, use **\1**. This will replace the entire found text of **Who ** with Who , which still includes that space. Perform this operation three times, and the new HTML looks like:

Who steals my purse steals trash;
'tis something, nothing;

Example 2: Cleaned up HTML.

Much cleaner HTML.

Making a Regex Find and Replace

Flare supports regex for both find and replace, and single files and multiple files.

To use regex in Flare for a single file:

1. Within a Flare project, open a topic file, paste the Example 1 text into the body of the HTML in the Text Editor view.
2. Open **Quick Replace** (Home|Find and Replace|Quick Replace, or Ctrl-H). The Find window opens in the top right of the topic page.
3. Enter **(.*?)<\/span>** in the Find textbox.
4. In the **Filter Options** button in the Find window, select **Regular Expressions**.
5. Click the **Find Next** arrow in the Find window to select the text. This highlights the found text. While this step isn't technically necessary for a find and replace option, it does confirm the find portion.
6. Click Replace Next in the Find window. This makes the replacement and highlights the next find text.
7. To speed this up, click **Replace All**. This replaces the remaining instances.

To use regex in Flare for multiple files:

1. Within a Flare project, paste the Example 1 text into the body of at least two HTML files in the Text Editor view.
2. Open **Find and Replace** in **Files** (Home|Find and Replace| Find and Replace in Files, or Ctrl-Shift-F). The Find window opens in the top right of the topic page. The Find and Replace in Files panel opens to the right.
3. Enter **(.*?)<\/span>** in the Find textbox.
4. Enter \1 in the Replace With textbox.
5. Check **Find** in **Source Code**.
6. Select **Regular Expressions** from Search Type.
7. Click **Find Next**. This highlights the next found text.
8. Click **Replace** to make the selection.
9. To speed this up, click **Replace All**. This replaces the remaining instances.

Other tools can make these changes too. One example is NotePad++ (https://notepad-plus-plus.org/). This is an improved version of the venerable Windows NotePad but adds dozens of additional features, including regex. This is not a unique

product and to be clear there are plenty of similar ones. To use regex in NotePad++:

1. Within a Flare project, paste the Example 1 text into the body of an HTML file in the Text Editor view.
2. Launch NotePad++.
3. Paste the Example 1 text into the body of the HTML into the default open document, likely named New1.
4. To better the see text, you may use line wrapping by selecting **View|Wrap**.
5. Open the Find dialog by clicking Ctl-F.
6. Select the Replace tab.
7. Select Regular expression in the Search Mode panel.
8. Enter in **Find What**: (.*?)<\/span>
9. Enter in **Replace** with: \1
10. Click **Find Next**. The first match will get highlighted.
11. Click **Replace**. The replaced text no longer has the span tags.
12. To speed this up, click **Replace All**. This replaces the remaining instances.

To do this for multiple pages:

1. Within a Flare project, paste the Example 1 text into the body of at least two HTML files in the Text Editor view.
2. Click **Find in Files**.
3. Make sure In all sub-folders is checked.
4. In the Directory textbox, enter a path, such as D:\Connectors\NewProject1\Content, or click the ellipsis button (…) to navigate to the path.
5. Click **Find Al**l. This displays all the instances found along the path.
6. This next step is the scary one. Be certain that the replacement is correct for all the instances. While typically you have to save each changed file, when making multiple instance changes, the replacements are saved automatically. You would be correct in thinking this could be a dangerous action, and it is. If you're connected to a source control system like Git, check in, push and pull all the files first before making global changes. That way you can restore the entire project should a mistake be made.
7. Click **Replace in Files** to make all the replacements. Don't worry yet, you'll be asked for confirmation.

Final Thoughts

Regex is a powerful and versatile find and replace tool. Because it is different than conventional find and replace, it requires thinking a little differently, that of patterns and not text. It is a rich language with nuisance. That means two things. First, you almost never want to read someone else's expressions. Depending how they craft them, it might be difficult to decipher. For example, **[a-z0-9_\.-]+@[\da-z\.-]+\.[a-z\.]{2,6}** matches any e-mail address, although it may take a moment to figure out why. Second, it is crafting. Writing the correct expression is tricky and at first time consuming. I recommend using an online regex tester, which shows the results of the searches in real time, to build your expression little by little. Then paste that expression into your application.

This is not a tool to avoid or to be afraid of. I have shown with the samples above that a short expression can do a lot. And with multipage options you can accomplish even more in less time. I feel when you start seeing the value of this, you'll also start seeing opportunities to use them.

Author Contact Information

Robert Delwood
Lead API Documentation Writer
WriteOnce.org
Robert@WriteOnce.org

Author Biography

Robert Delwood is a programmer, writer, and programmer-writer currently in Chicago but formerly with NASA's Johnson Space Center in Houston. He's passionate about technical writing, procedural, conceptual, and for API documentation. With more than 18 years' experience, he's written about and documented topics from Windows kernel-level device drivers and speech recognition APIs/SDKs for Microsoft, to help desk procedures and application manuals for the military.

He has two specializations: API documentation and Microsoft Office automation. He's documented APIs for more than a dozen years, and stresses the

importance of upping everyone's game from good documentation to great documentation. As a programmer-writer this includes writing original code samples, sample applications, and attention to detail in API reference pages.

For Microsoft Office automation he believes every writing team can use a programmer-writer. Customized tools can streamline any work process, in instances from weeks to literally minutes. It can also improve quality and perform task considered impossible by manual means. This can be accomplished internally, without involving the development department.

He's authored three books. The newest is about writing great API documentation and is due out in summer of 2018. *The Secret Life of Word* (XML Press, http://xmlpress.net) is about Word's automation for technical writers, non-programmers, knowledge workers, or anyone who wants to do more things quickly with Word.