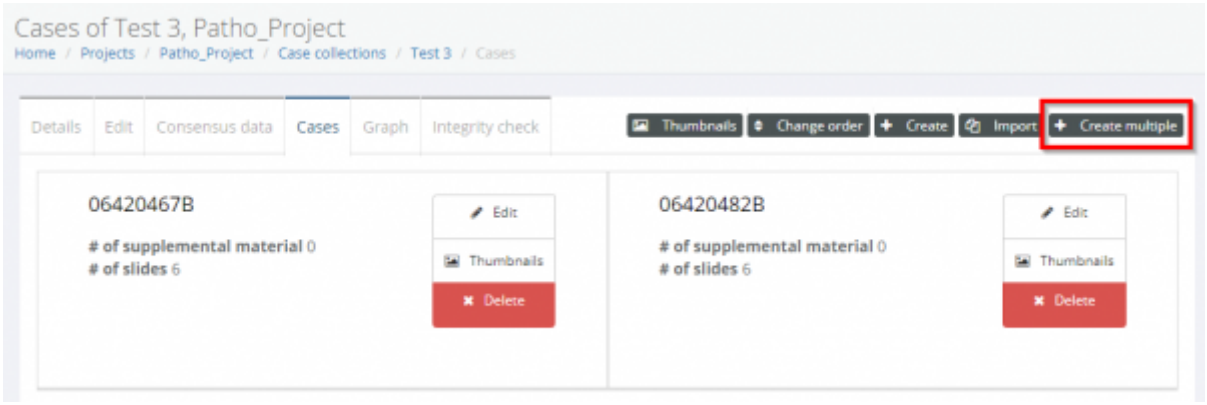
Creating (Multiple) Cases with Pattern Search

Sometimes you may need to create many Cases from WSIs that aren't in their own subfolder, or create a Case that has slides across several different folders, or find a large number of examples of a particular stain, indication, or filetype.

That's where Pattern Search can make your life a lot easier by eliminating the manual searching through all your directories to add them one by one.

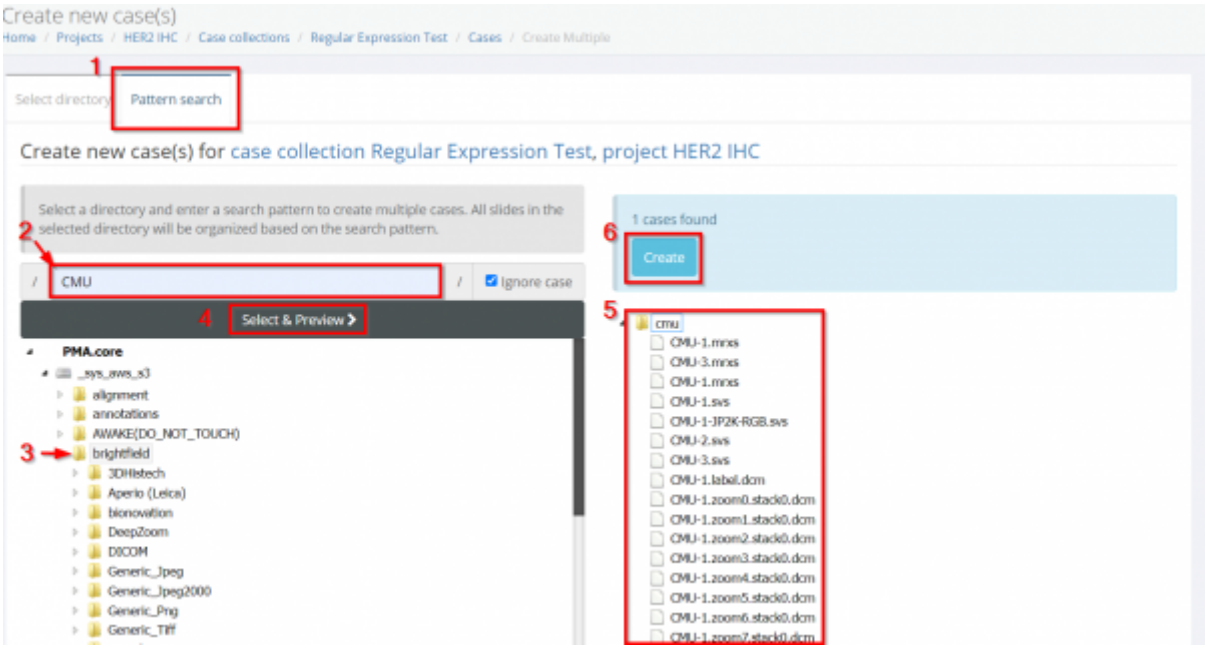
If you're not sure how to get to the Cases menu pictured below, first read Create Cases and Case Collections, then come back here.

Then click the “Create Multiple” button.



Running a pattern search

1. First click the 'Pattern Search' tab at the top of the page
2. Type in your search term, this is the text you will be searching for. The pattern search follows 'regular expression' rules. More information on how to run these is at the bottom of this page
3. Third select the directory you want to run your search in. The pattern search will be applied to every file in every (sub)folder
4. Click 'Select and Preview' to run the search
5. Expand the folder view to see the files your search has yielded
6. When you're happy with the results, click Create. If you want to add/remove files you can edit the case after creation.



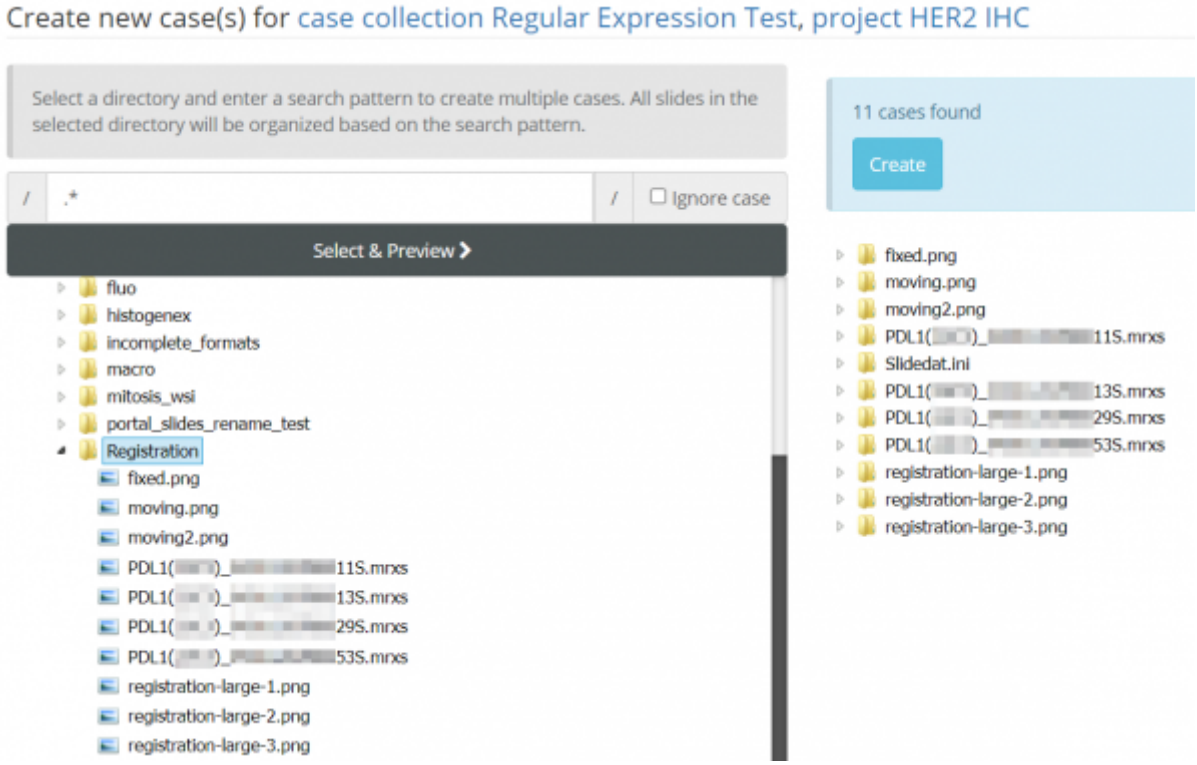
Pattern Search Behaviour and how to use 'Regular Expression' in a Search

Regular expressions, often abbreviated as regex or regexp, are powerful tools for pattern matching and text manipulation. They provide a concise and flexible way to describe, search, and manipulate strings.

Adding Every File as it's Own Case

.*

That's all you need. The '.' matches any character except for new line, and the '*' says the preceeding character can be matched 0 or more times.
In the screenshot below, you can see this search makes a new case for every WSI file in the directory.

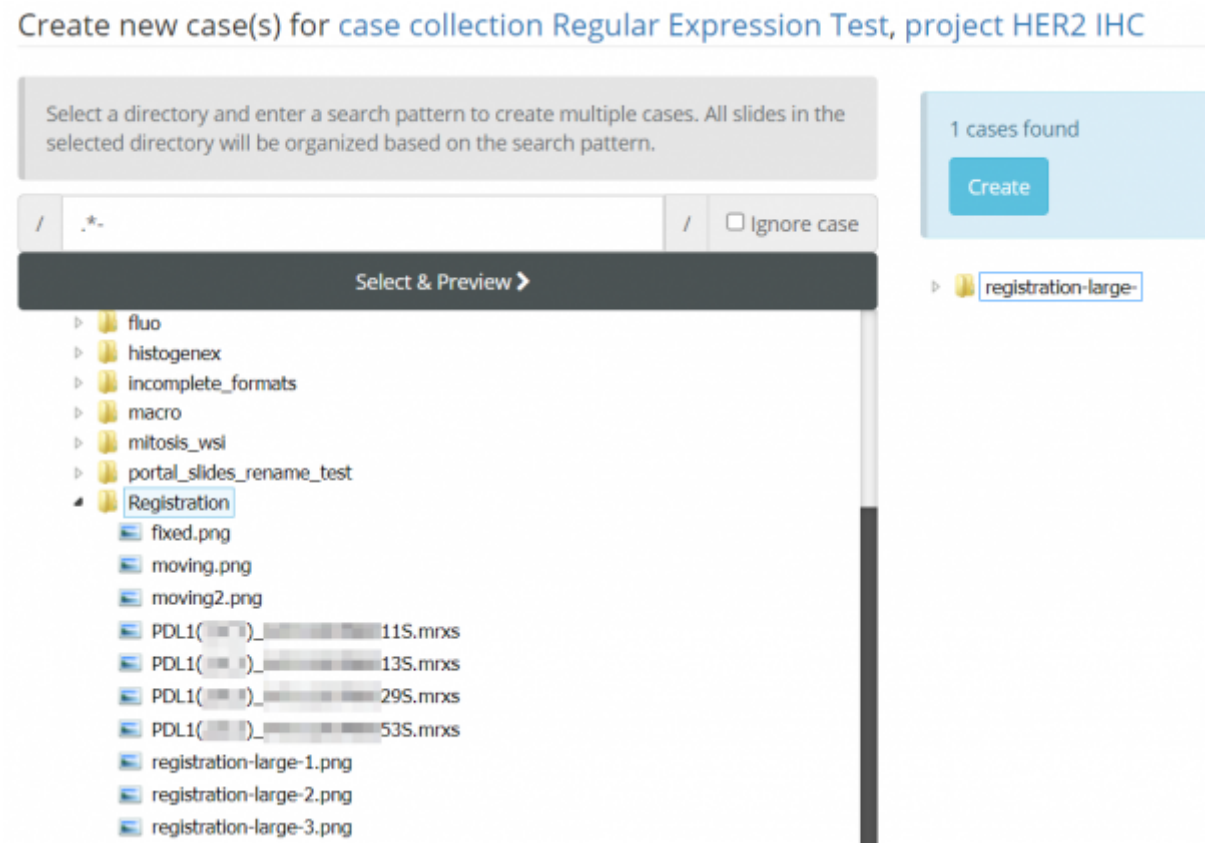


Here are some examples of how .* behaves in a regex pattern:

abc.*def: This pattern would match strings that start with “abc” and are followed by any sequence of characters, eventually ending with “def”. For example, it would match “abc123def” or “abcxyzdef”.

.*\d+.*: This pattern would match strings that contain one or more digits anywhere in the string. The .* before and after \d+ allows for any characters before and after the digit(s).

In the screenshot below you can see that '.*-' matches only files that have a '-' in their name

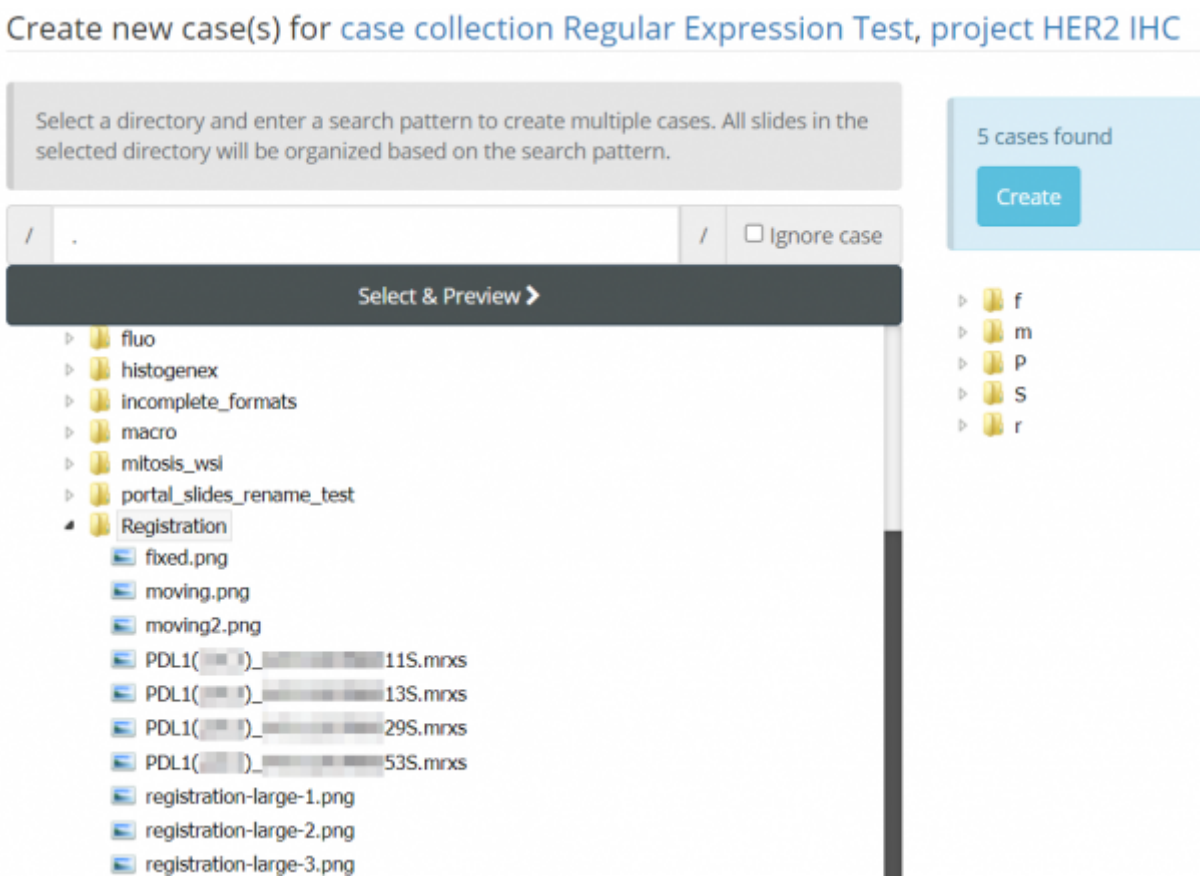


Adding Similarly Named Files into Cases

What about if you have file(s) named in a similar way?

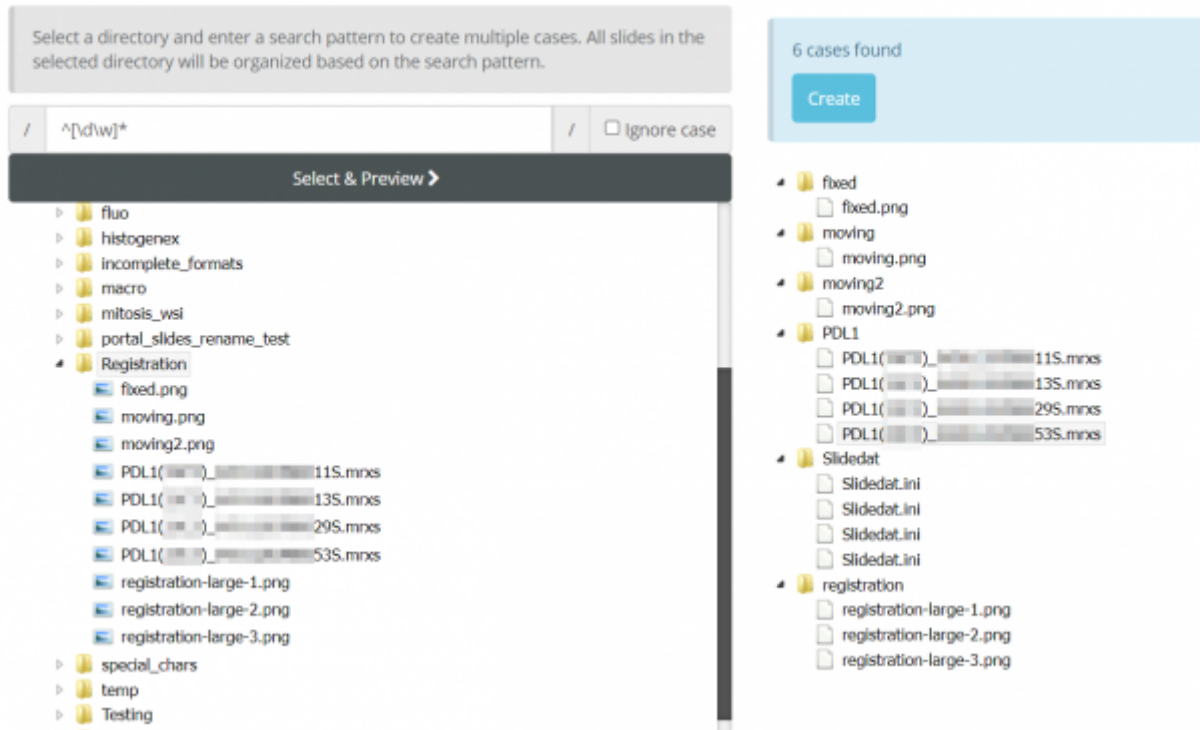
We can use some simple commands to sort these into cases.

Here you can see that typing '.' will match all files starting with the same character into a Case.



If you had files with different names, but each case had the first 4 letters or numbers, you could type '....' to group them together.

Another way to search and group together files into Cases is using [].
Any search terms you put within the [] will be searched for.
In the example below, we use ^[\d\w]* to find the starting pattern of each filename until it is interrupted by a special character e.g. '(' or '.'



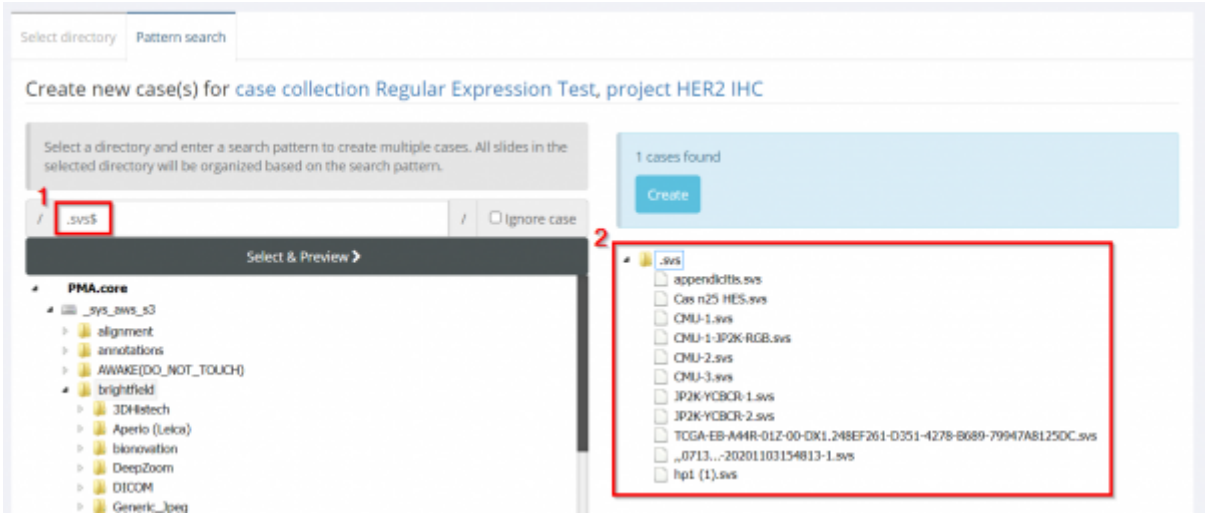
Finding Many Files for a Single Case

You can use pattern search like a regular search engine to find 'files' in your selected folder

Some examples which use several of these blocks together, for example:

Using $ at the end of your search term to match the end of the filename, e.g. .filetype_name$ will only include files ending in .filetype_name
Using ^CF to match any string starting with 'CF'
Using ? to indicate subexpressions in your pattern, e.g. “fish(er)?” matches “fish” or “fisher”.

In the screenshot below you can see the $ regex [1] has been used to select only .svs files across many subfolders [2]



Full Explanation of Regular Expression Syntax

Here are some key concepts and syntax:

1. Literal Characters:

Most characters in a regular expression match themselves literally.
For example, the regular expression abc will match the string “abc” in the input.

2. Metacharacters:

```
• Certain characters have special meanings in regular expressions and are
called metacharacters. Some common ones include:
    ○ . (dot): Matches any single character except a newline.
    ○ *: Matches 0 or more occurrences of the preceding character or group.
    ○ +: Matches 1 or more occurrences of the preceding character or group.
    ○ ?: Matches 0 or 1 occurrence of the preceding character or group.
    ○ ^: Anchors the regex at the beginning of the line.
    ○ $: Anchors the regex at the end of the line.
```

3. Character Classes:

```
• [ ]: Defines a character class. For example, [aeiou] matches any vowel.
• [^ ]: Negates a character class. For example, [^0-9] matches any non-digit
character.
```

4. Quantifiers:

```
• {n}: Matches exactly n occurrences of the preceding character or group.
• {n,}: Matches n or more occurrences.
• {n,m}: Matches between n and m occurrences.
```

5. Grouping and Capturing:

```
• ( ): Groups expressions together. Also used for capturing portions of the
matched text.
```

6. Anchors:

```
• \b: Word boundary anchor.
• \B: Non-word boundary anchor.
• ^ and $: Anchors for the start and end of a line, respectively.
```

7. Escape Characters:

```
• \: Escapes a metacharacter, allowing it to be treated as a literal
character. For example, \. matches a literal dot.
```

8. Modifiers:

```
• i: Case-insensitive matching.
• g: Global matching (find all matches rather than stopping after the first
match).
```

9. Special Sequences:

```
• \d: Matches any digit (equivalent to [0-9]).
• \w: Matches any word character (alphanumeric + underscore).
• \s: Matches any whitespace character.
```

10. Lookaheads and Lookbehinds:

```
• (?=...): Positive lookahead assertion.
• (?!...): Negative lookahead assertion.
• (?<=...): Positive lookbehind assertion.
• (?<!...): Negative lookbehind assertion.
```

Regular expressions are supported in many programming languages (e.g., Python, JavaScript, Java) and text editors. They are incredibly versatile but can also become complex. Practice and experimentation are key to mastering regex. Online regex testers, such as regex101 or RegExr, can help you test and visualize your regular expressions.