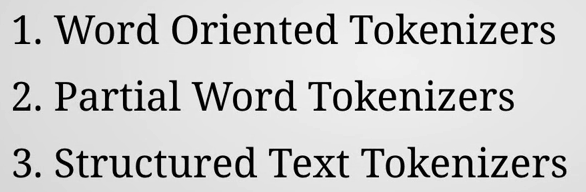
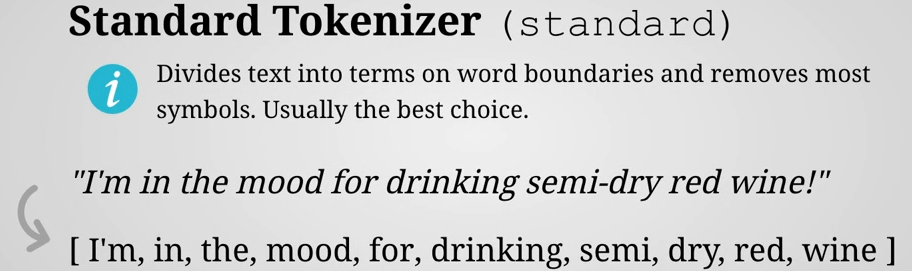
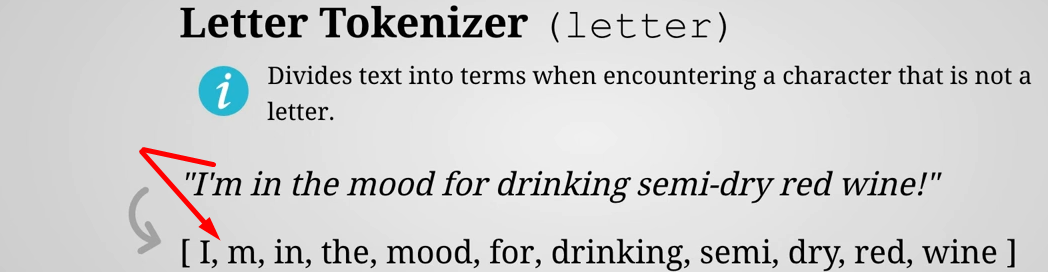
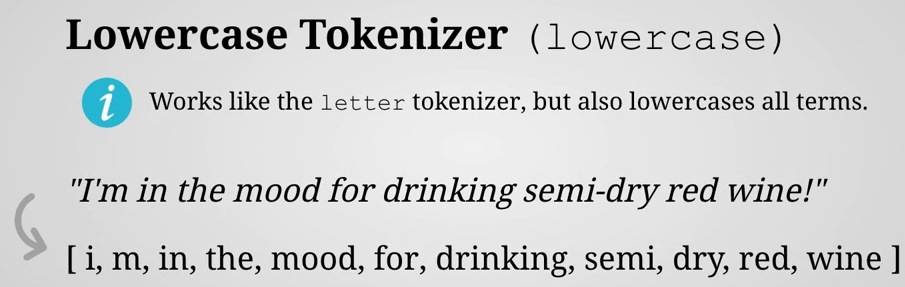
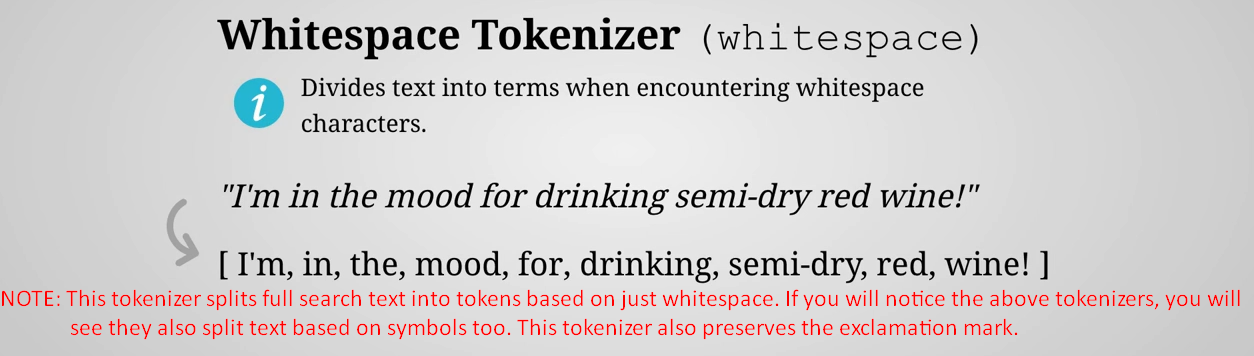
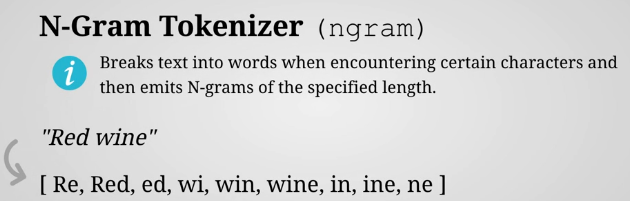
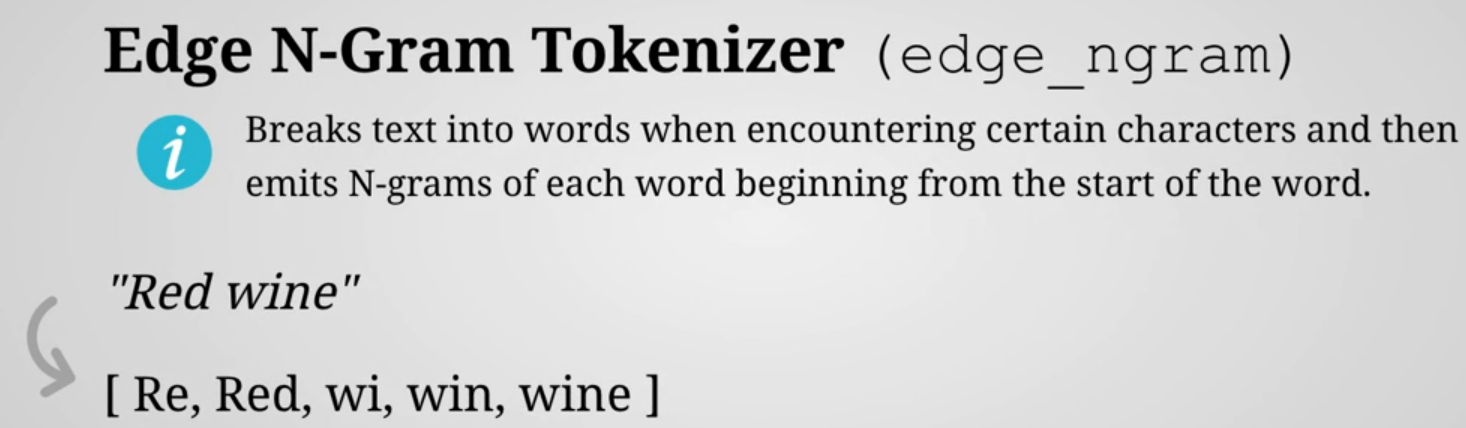
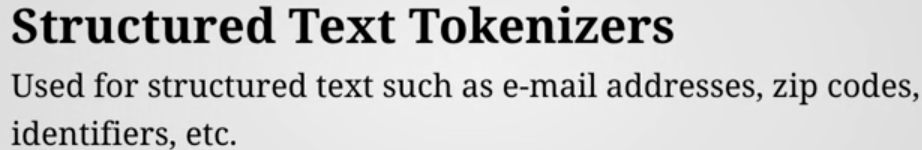
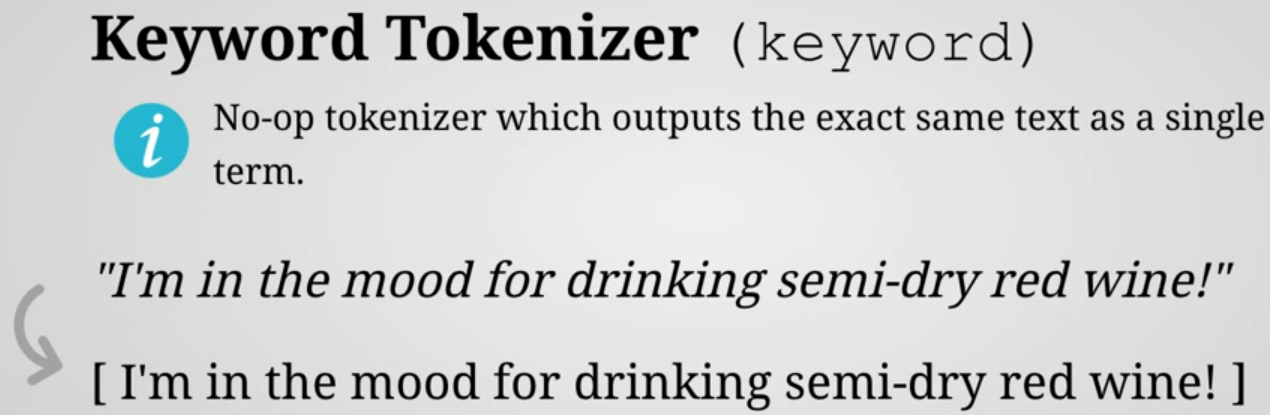
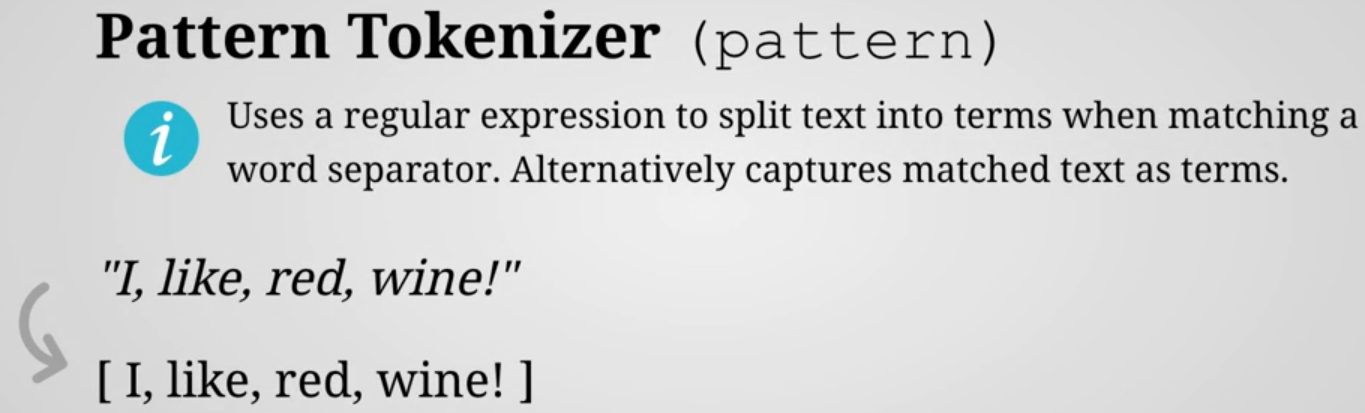
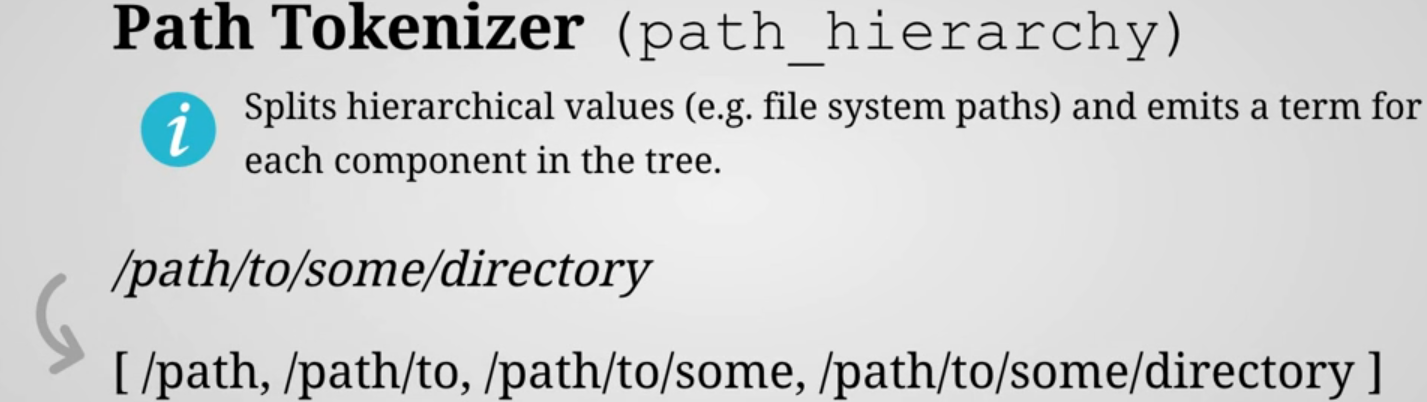
1. 
2. **Agenda:**
   1. Go through a number of tokenizers that Elasticsearch provides by default.
3. You have already seen how to apply tokenizer with the analyzer API.
4. Tokenizers are grouped into 3 main categories.  
   
   1. **Word Oriented Tokenizers**:  
      
      1. This group of tokenizers are used to tokenize full text into words.
      2. These tokenizers work at the word level so the output will be easily readable by humans.
      3. Examples for Word Oriented Tokenizers.
         1. **Standard Tokenizer** (standard).  
              
            This tokenizer splits text into terms based on white spaces and certain symbols.  
            It also removes certain symbols which will not be part of the output. Such as hyphen (-) as you can see snapshot.
         2. **Letter Tokenizer** (letter)  
            ****This tokenizer is more primitive which may be useful in some cases.
         3. **Lowercase Tokenizer** (lowercase)  
            ****
         4. **Whitespace Tokenizer** (whitespace)  
            ****
         5. **UAX URL Email Tokenizer** (uax\_url\_email)  
            ****
   2. **Partial Word Tokenizers**
      1. ****
      2. **Example:**
         1. **N-Gram Tokenizer** (ngram) Breaks the text into words similar to the standard tokenizer and then emit so-called **n-gram** of a specified length.   
            
            1. It’s like a sliding window that moves across a word and includes a number of characters. You can kind of think of it as the substring method used in programming language with an increasing start index.
            2. The number of characters included in n-gram depends on how we configure it.
            3. As you can see here I’ve used a minimum number of characters of two and a maximum number of 10 meanings that each n-gram will be b/w 2 and 10 characters in length.
            4. Let’s look a little closer at the example.
            5. The text is “Red wine”.
            6. So first the tokenizer splits the text into two words “Red”, “wine” based on whitespace.
            7. It then begins by generating n-gram’s for the first term being the word “red”.
            8. It takes the first two characters because this is what was configured as the minimum number of characters that should be included.
            9. Then it simply increases the number of characters and keeps on increasing until it either reaches the end of the term or reaches the maximum character account.   
               So, in our case the maximum count is set to 10 and the length of the term is free. So it reaches the end of the term  
               When that happens it moves the cursor so to speak or start index one position forward and starts generating n-grams from position of the “e” character that generates an “**ed**”whichis the last n-gram for the term.
            10. That’s because we have a minimum character count of two, so moving the sliding window one position forward will not result in any new n-grams then tokenizer moves onto the next term being “wine” and does exactly the same thing.
         2. **Edge N-Gram Tokenizer (edge\_ngram)  
            **
            1. It does the same thing as what you’ve just seen except that it generates n-grams from the being of the terms.  
               Comparing with substring() in programming languages, the start index always remains at the zero.
            2. Edge n-gram tokenizer is often used for **auto-completion**. Similar to what you see in Google for example.  
               Performance-wise it is not better as we have something called **suggesters.**
         3. **Structured Text Tokenizers**:  
            
            1. It is used basically for any structured text.
         4. **Keyword Tokenizer** (keyword)
            1. ****
            2. It doesn’t do anything. It takes the input and returns it as a single token.
            3. Since analyzers require one tokenizer to be specified. This can be useful when you don’t want to tokenizer.
         5. ***P*attern Tokenizer** (pattern)
            1. ****
            2. Pattern tokenizer uses a regular expression to do one of two things either matches token separators and uses those to split text into terms or use capturing groups and use the matched text as terms.  
               In the above, words are separated by commas in the inputs.  
               The regular express used is simply consisting of comma but could be a more complex one as well.
            3. The expression matches the tokens separators and the matching parts are stripped from the input and the text is split into tokens at those positions. The result is therefore tokens without the commas split at the positions of the commas.  
               Notice that the commas are not preserved and that the commas in the output are just for separating the tokens. They are not part of the tokens themselves.
         6. **Path Tokenizer** (path\_hierarchy)
            1. 
            2. It splits the hierarchical values such as file system path by a path separator and emits a token for each component.
            3. This tokenizer is used when the input is a **tree structure** which is the case for File System Path for instance.
            4. **NOTE**: We can configure the delimiter that is used to split the input but it defaults to a forward slash.  
               You can also specify a replacement value which each occurrence of the delimiter will be replaced with.  
               The value of the delimiter is optional by default.
            5. So in the example that I’ve used you can see how a UNIX file system path is split by forward slashes and how a term is output for each level in the tree structure and that’s it.