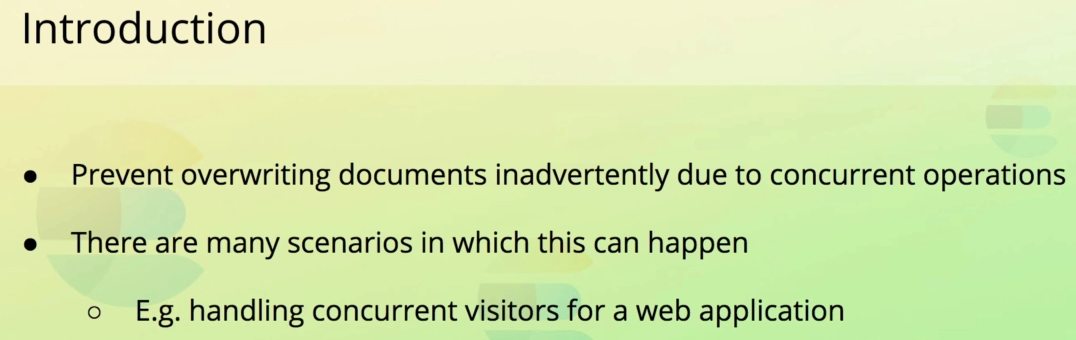
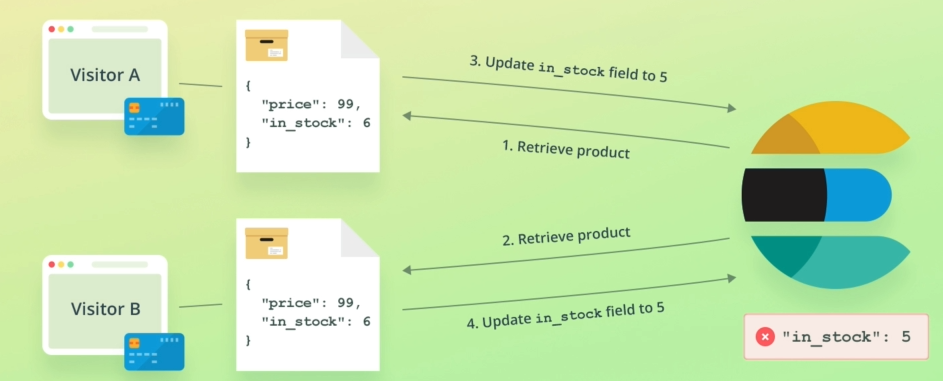
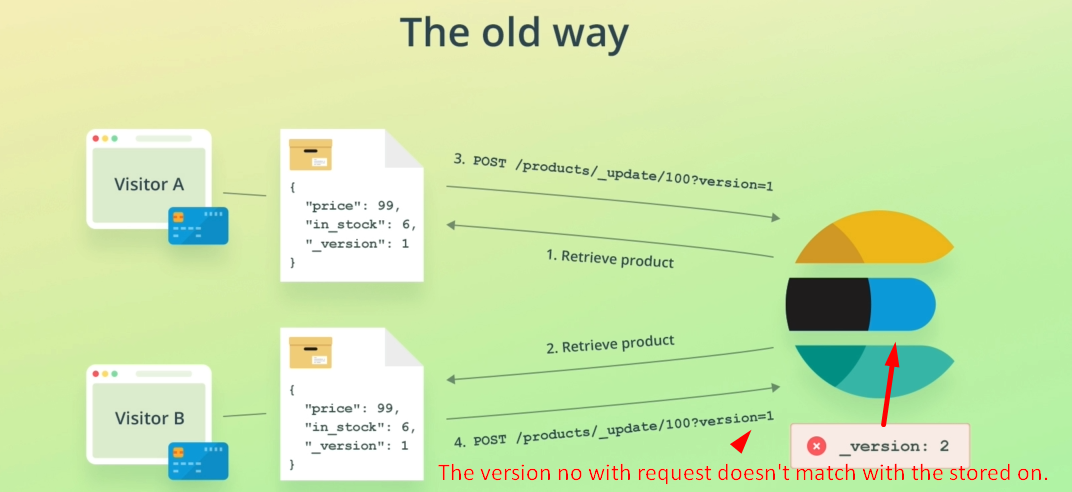
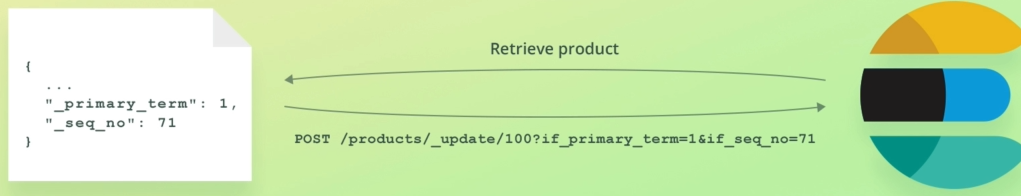
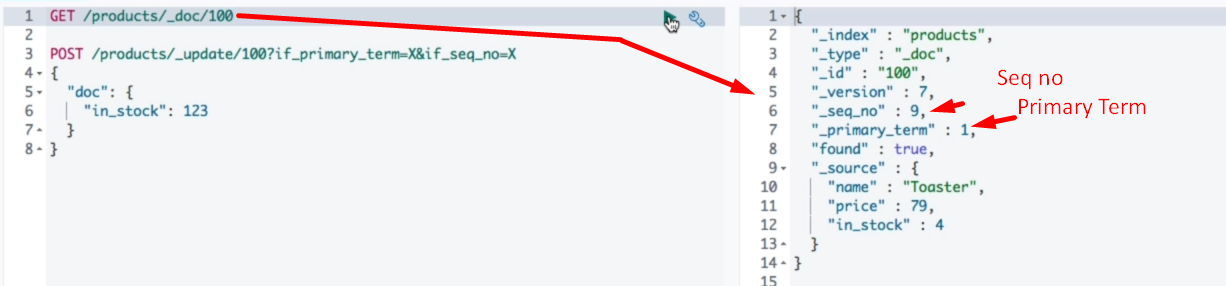
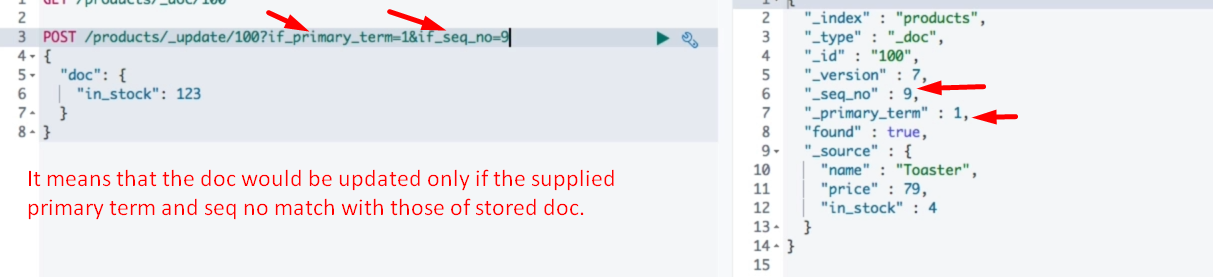
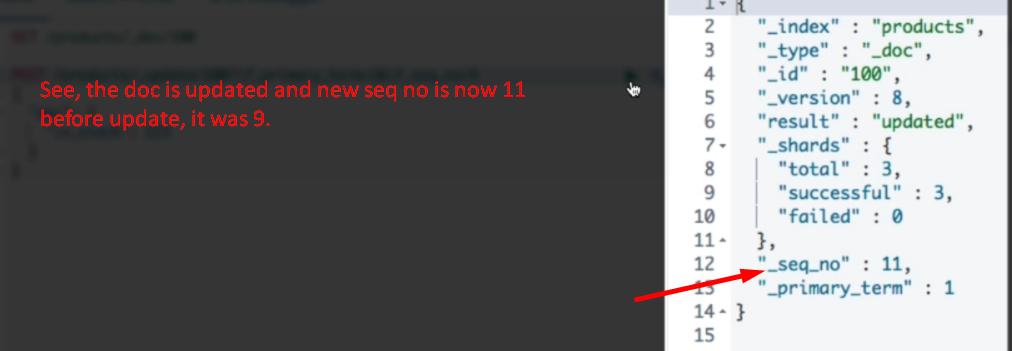
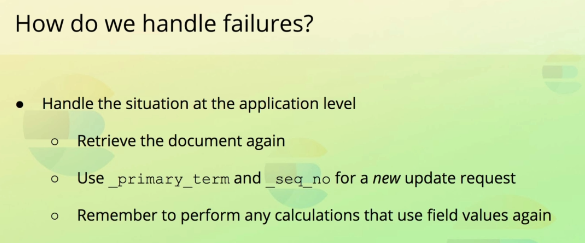
1. 
2. **Agenda**: We will talk about Optimistic Concurrency Control. This is essentially a way to prevent that an old version of a document overwrites a more recent one, i.e. if write operations arrive out of sequence. Since Elasticsearch is distributed and it involves networking, such a scenario can occur.  
   There are quite a few examples of when the ordering of write operations may be incorrect, but let’s go with a simple example:  
   
3. **Example**: When a web application updates a field more specifically, an ecommerce application. Suppose that a website visitor has added an item to the cart, and completed the checkout flow. Once that happens, the application retrieves the product from Elasticsearch. At this particular point in time, another visitor completes the checkout flow for the same product, and a different thread on the web server also retrieving the product. At this point in time, both threads have retrieved the same product. The first thread subtracts the product’s ïn\_stock”field by one and updates the product through the Elasticsearch API.  
   The second thread does the same thing, and this is where we run into trouble. The second thread thinks it has the latest value for the ïn\_stock”field but it has been updated from six to five since the product was retrieved. One is subtracted from the incorrect value, and the product is updated again, but with the same field value. There are no errors indicating that anything went wrong, and the application believes everything isokay. The ïn\_stock”field, however now has a value of five, where the value should have been 4. This consequence of this varies based on the application, but in this case we could be selling products that are not in stock, resulting in a bad customer experience.   
   
4. **So, how do we prevent this from happening**?  
   We need our update to fail if the document has been modified since we retrieved it. This is where versioning comes in. Well at least it used to.
5. The old way of accomplishing this was to use the “\_version” field returned by the document retrieval and send that along with the update request as a query parameter. The update operation would then fail if the supplied didn’t match the one stored within the index.  
   That approach worked well in most cases, but it had some flaws when things went wrong pretty much the problems that primary terms and sequence numbers solve.  
   Therefore the new approach is to use primary terms and sequence numbers instead. Let’s walk through an example.  
   
6. When we retrieve the product, the current primary term and sequence number are included within the results.  
   I will show you these fields in just a moment. What we can do, is to take the values and add them to the POST request that we send to update the document. To do that, we use the **ïf\_seq\_no**”and “**if\_primary\_term**”parameters. Elasticsearch will then use these two values to ensure that we won’t overwrite a document inadvertently if it has changed since we retrieved it. If that happens, the operation will fail, and we can try the process again.  
     
   Let’s go back to Kibana and see that in action.  
   We will retrieve a doc because we want to see the the primary term and sequence no within the result.
7. 
8. 
9. 
10. 
11. Okey, so we’re not able to overwrite field values if a document has been updated without knowledge.
12. **But how do we proceed when getting this error, then?**
13.   
     When updating documents this way, you will need to handle this within your application meaning that you should retrieve the document again, and use the new primary term and sequence when retrying the update. Remember to perform any calculations again since field values might have changed.
14. 