Design Patterns

Builder Pattern

```
   /**
    * Builder class for {@link IndexResponse}. This builder is usually used during
xcontent parsing to
    * temporarily store the parsed values, then the {@link Builder#build()} method
is called to
    * instantiate the {@link IndexResponse}.
    */
   public static class Builder extends DocWriteResponse.Builder {
       @Override
       public IndexResponse build() {
           IndexResponse indexResponse = new IndexResponse(shardId, id, seqNo,
primaryTerm, version, result);
           indexResponse.setForcedRefresh(forcedRefresh);
           if (shardInfo != null) {
               indexResponse.setShardInfo(shardInfo);
           }
           return indexResponse;
       }
   }
}
```

We found an example of Builder Pattern in the file org/elasticsearch/action/index/IndexResponse.java.
The builder class is created for the IndexResponse class. It provides a method called **build** to create an instance of IndexResponse. As described in the commentation, the builder is used when we are in the process of xcontent parsing and we need to store the parsed values provisionally.

Factory Pattern

Factory Pattern provides an interface for the subclasses to decide which class to instantiate. In the file org/elasticsearch/common/xcontent/XContentFactory.java, we found that it provides a method to construct different builders according to the given content type, which is a typical example of the factory pattern.

```
/**
* Constructs a xcontent builder that will output the result into the provided
output stream.
*/
public static XContentBuilder contentBuilder(XContentType type, OutputStream
outputStream) throws IOException {
   if (type == XContentType.JSON) {
       return jsonBuilder(outputStream);
   } else if (type == XContentType.SMILE) {
       return smileBuilder(outputStream);
   } else if (type == XContentType.YAML) {
       return yamlBuilder(outputStream);
   } else if (type == XContentType.CBOR) {
       return cborBuilder(outputStream);
   }
   throw new IllegalArgumentException("No matching content type for " + type);
}
```

Strategy Pattern

Strategy pattern is a behavioral software design pattern. It defines an algorithm and multiple implementation, it allows the program to select the appropriate algorithm in the run time.
In our project, There is an interface **NodeSelector**, it has a method select(). NodeSelector has multiple implementations such as **HasAttributeNodeSelector** and **PreferHasAttributeNodeSelector**. In the class RestClientBuilder(which implements the builder pattern), there is a setter **setNodeSelector** takes in a NodeSelector as parameter.

```
public RestClientBuilder setNodeSelector(NodeSelector nodeSelector) {
   Objects.requireNonNull(nodeSelector, "nodeSelector must not be null");
   this.nodeSelector = nodeSelector;
   return this;
}
```

When you need to build a RestClientBuilder and set its NodeSelector, you can pass in different kinds of nodeSelector strategically for different purposes.

Iterator Pattern

Iterator Pattern provides a method to iterate through different data structures or collection objects. It typically has a hasNext() method and a next() method. The hasNext() method is used to check whether the data object has the next element to be accessed. The next() method is

used to fetch the next data element and return the data. Here's an example of our project. It implements the Iterator interface and overrides the typical hasNext() method and next() mothed.

```
static class ConcatenatedIterator<T> implements Iterator<T> {
   private final Iterator<? extends T>[] iterators;
   private int index = 0;

   ConcatenatedIterator(Iterator<? extends T>... iterators) {
       if (iterators == null) {
           throw new NullPointerException("iterators");
       }
       for (int i = 0; i < iterators.length; i++) {
           if (iterators[i] == null) {
               throw new NullPointerException("iterators[" + i  + "]");
           }
       }
       this.iterators = iterators;
   }

   @Override
   public boolean hasNext() {
       boolean hasNext = false;
       while (index < iterators.length && !(hasNext = iterators[index].hasNext())) {
           index++;
       }

       return hasNext;
   }

   @Override
   public T next() {
       if (!hasNext()) {
           throw new NoSuchElementException();
       }
       return iterators[index].next();
   }
}
```

Template Method Pattern

A template method means to define a skeleton of the function behavior and leave the implementation details to the child classes. For example, in our project, there is an abstract class named *Query*. According to the comments, this class is used to represent different queries.

Here are part of the abstract class's codes. It defines the skeletons of some methods such as *containsNestedField()*, *addNestedField()*, and *enrichNestedSort()* etc.

```
/**
* Intermediate representation of queries that is rewritten to fetch
* otherwise unreferenced nested fields and then used to build
* Elasticsearch {@link QueryBuilder}s.
*/
public abstract class Query {
   private final Source source;

   Query(Source source) {
       if (source == null) {
           throw new IllegalArgumentException("location must be
specified");
       }
       this.source = source;
   }

   /**
    * Location in the source statement.
    */
   public Source source() {
       return source;
   }

   /**
    * Does this query contain a particular nested field?
    */
   public abstract boolean containsNestedField(String path, String field);

.....
```

The child clssses of *Query* implement different details of these methods. For example, the class *BoolQuery* extends the abstract class. In this class, it represents the boolean AND or boolean OR. It overrides the abstract methods mentioned above and implements the details of the class's behaviors.

```
/**
* Query representing boolean AND or boolean OR.
*/
public class BoolQuery extends Query {
```

```
   /**
    * {@code true} for boolean {@code AND}, {@code false} for boolean {@code OR}.
    */
   private final boolean isAnd;
   private final Query left;
   private final Query right;

   public BoolQuery(Source source, boolean isAnd, Query left, Query right) {
       super(source);
       if (left == null) {
           throw new IllegalArgumentException("left is required");
       }
       if (right == null) {
           throw new IllegalArgumentException("right is required");
       }
       this.isAnd = isAnd;
       this.left = left;
       this.right = right;
   }

   @Override
   public boolean containsNestedField(String path, String field) {
       return left.containsNestedField(path, field) || right.containsNestedField(path,
field);
   }

   @Override
   public Query addNestedField(String path, String field, String format, boolean
hasDocValues) {
       Query rewrittenLeft = left.addNestedField(path, field, format, hasDocValues);
       Query rewrittenRight = right.addNestedField(path, field, format, hasDocValues);
       if (rewrittenLeft == left && rewrittenRight == right) {
           return this;
       }
       return new BoolQuery(source(), isAnd, rewrittenLeft, rewrittenRight);
   }

   @Override
   public void enrichNestedSort(NestedSortBuilder sort) {
       left.enrichNestedSort(sort);
       right.enrichNestedSort(sort);
   }

   @Override
   public QueryBuilder asBuilder() {
       BoolQueryBuilder boolQuery = boolQuery();
       if (isAnd) {
           boolQuery.must(left.asBuilder());
           boolQuery.must(right.asBuilder());
       } else {
```

```
            boolQuery.should(left.asBuilder());
            boolQuery.should(right.asBuilder());
        }
        return boolQuery;
    }
```

First Issue

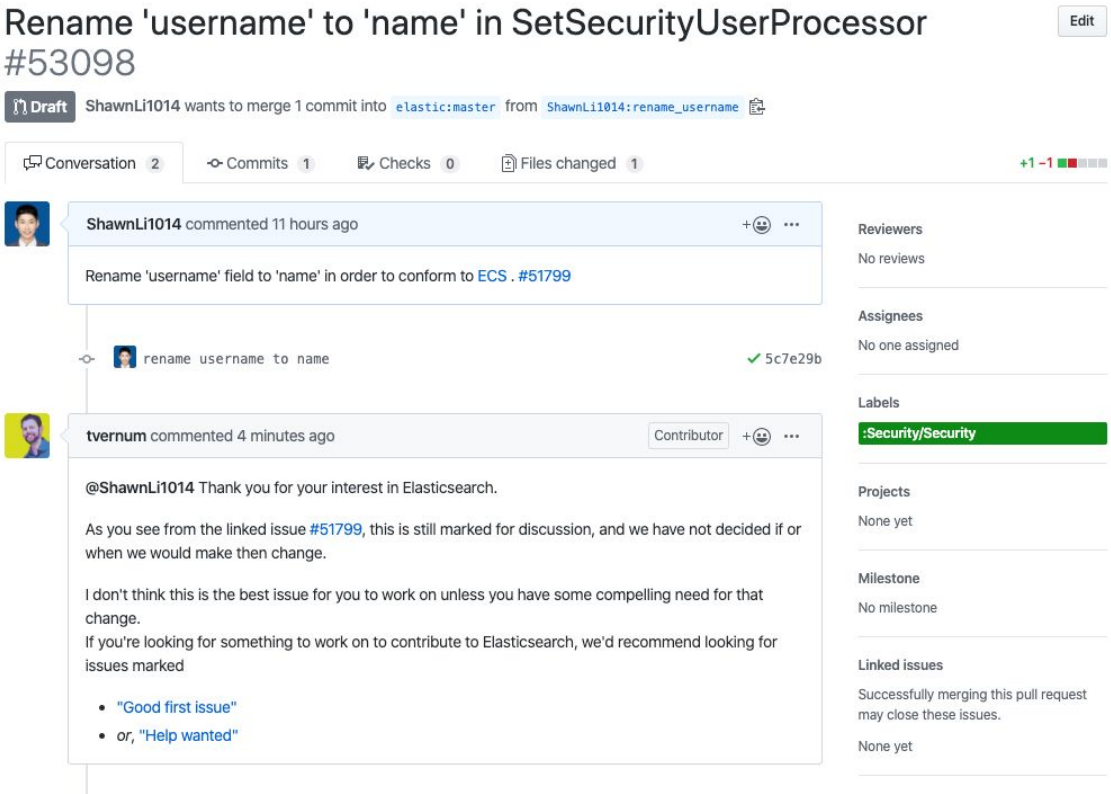
The issue we implemented is **Rename `user.username` to `user.name` for SetSecurityUserProcessor #51799.** This issue describes that the class SetSecurityUserProcessor auguments ingested documents by adding an user object, but the username field does not conform to the ECS(Elastic Common Schema). As described in ECS, the username field should be name.
So we changed

```
if (user.principal() != null) {
   userObject.put("username", user.principal());
}
```

To

```
if (user.principal() != null) {
   userObject.put("name", user.principal());
}
```

We submitted a pull request and received a response from the project owner.



Although our pull request didn’t get accepted, we really appreciate the reply from the elasticsearch team and they provide some other interesting issues for us to work on