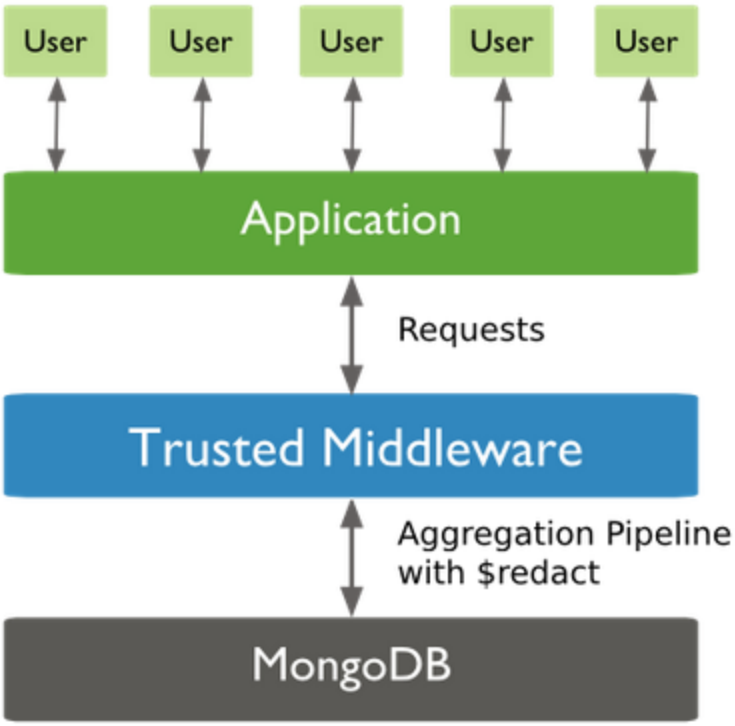
MongoDB Field-Level Access Control

- Introduction
- Security Markings
  - Security Markings Example
- Redaction Expression
  - Redaction Examples
- CAPCO Document Markings
  - CAPCO Document Markings Example
- CAPCO Redaction Expression
  - CAPCO Redaction Expression Example
- Example Implementation
  - Document/sub-document Markings
  - User Attributes
  - Redaction Expression
  - DBCollection Wrapper
- Appendix - Working Example
  - Step by Step

Introduction

Some users of MongoDB wish to restrict access to certain documents within a particular collection, or even to certain fields within a particular document. Frequently, these restrictions are a function of the content of the documents themselves, rather than of the schema of the document. Policies governing these restrictions may be simple or complex, and they vary between organizations. MongoDB now provides a flexible mechanism which facilitates applying these policies on queries. This mechanism/capability will be referred to as Field-Level Access Control (FLAC) in this document.

The approach taken to restrict read access to fields and documents is to combine queries that redact restricted fields and documents with end user queries over the data to which those end users (principals) are entitled. The arrangement of such an approach is as follows:

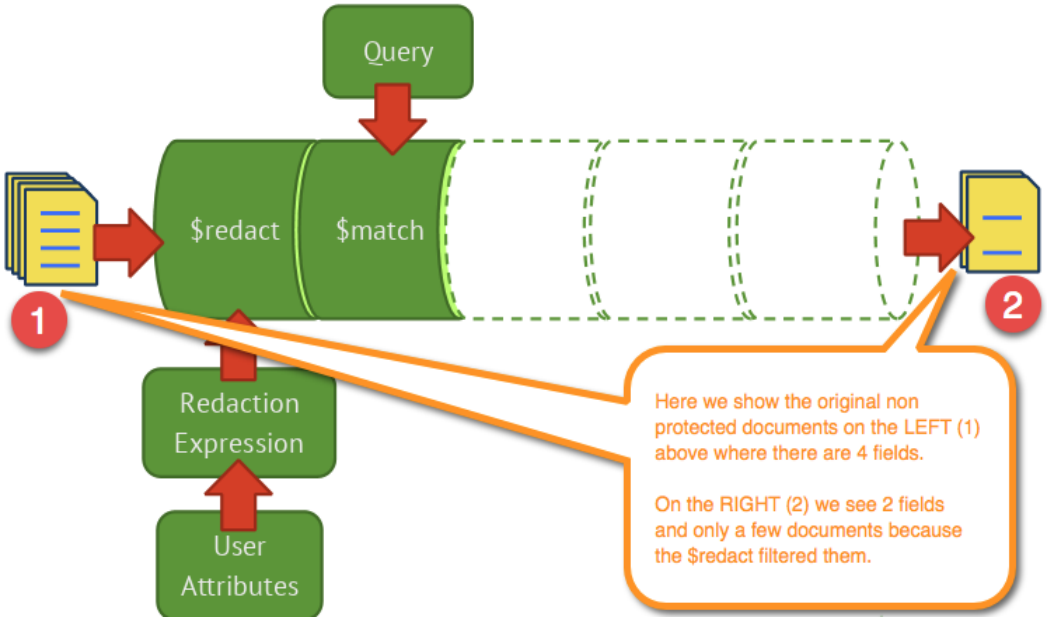


The construction of the redaction logic and combination with user attributes happens in the “Trusted Middleware” layer. The actual redaction however, is done in the database.

This is accomplished by leveraging the aggregation framework. FLAC uses aggregation and the special $redact aggregation framework pipeline stage. The $redact pipeline stage incorporates a redaction expression that represents the domain specific security logic for a particular application. This redaction expression relies upon User Attributes that tell the system what capability the current user has and evaluates against a set of user defined security markings within each document. See http://docs.mongodb.org/manual/core/aggregation-pipeline/ for additional details on the aggregation framework.

The $redact stage will limit visibility of documents and fields within the documents. The pipeline starts with the $redact stage to apply the field-level access control and the query and project arguments that might be given in a DBCollection.find(query, project) call are mapped into following stages in the aggregation pipeline. The aggregation execution engine will promote non-negating query terms that are specified in the $match stage to run before the $redact stage so that indexes can be leveraged. Specifying a $redact stage **after** a $match may make your pipeline vulnerable to data discovery via a process of comparing the complete data set to results of queries with negation terms ({ name : { $ne : “Joe” } }).

To illustrate this concept, consider the figure below. It shows the flow of how the original query and user attributes lead into a FLAC pipeline. The $redact expression and specific user attributes are used to generate the $redact stage and the query is used to generate a $match stage. Subsequent stages such as $sort and $limit can be added to further manipulate the output.



Security Markings

In order to control access to a entire document or sub-documents, each document should contain one or more security markings at each field/level within the document for fields (or sub-documents) that requires access control. These security markings can essentially be any JSON type, these markings would typically be represented using a string, integer, array of strings, or an array of integers. In the reference implementation, strings are used to represent Clearance, SCI programs, and Citizenship.

Security Markings Example

Consider the following example document with security markings:

```
{
   _id: 1,
   title: "123 Department Report",
   tags: [ "low" ],
   year: 2014,
   subsections: [
```

```
        {
            subtitle: "Section 1: Overview",
            tags: [ "low" ],
            content:  "Section 1 Content..."
        },
        {
            subtitle: "Section 2: Analysis",
             tags: [ "medium" ],
            content: "Section 2 Content..."
        },
        {
            subtitle: "Section 3: Budgeting",
            tags: [ "high" ],
            content: "Section 3 Content..."
        }
    ]
}
```

In this example, the field named `tags` is used as the security markings that will be used by the redaction expression. The security markings in this example is implemented as an array of strings. The `tags` field is included at the root level of the JSON document, and also at various sub-document levels. For the security marking values, a simple enumeration of the values "`high`", "`medium`", and "`low`" can be used to represent the various security levels required to access each field.

Redaction Expression

The redaction expression is used by the $redact pipeline stage to restrict contents of the result set based on the access required to view the data. The redaction expression conditionally returns one of "`$$KEEP`", "`$$PRUNE`", or "`$$DESCEND`". For each document, the redact stage performs a pre-order traversal of the input document tree, to construct an output document tree. For each node, it binds the node to the current path context and evaluates the redact expression. If the expression returns "`$$DESCEND`", it inserts a corresponding node in the output document and continues the traversals of the node's children. If it returns "`$$PRUNE`", it puts no node in the output document, and continues the traversal of the node's parent in the input document. If it returns "`$$KEEP`", it inserts the node and the node's children into the output document. Our reference example's redact expression only uses the $$PRUNE and $$DESCEND options.

Redaction Examples

Using the security markings example above, a simple redaction expression can be constructed and applied to an aggregation pipeline using the $redact pipeline stage and and can be executed via the `mongo` shell, so you can easily follow along:

```
var userAccess = [ "low" ];
db.report.aggregate( [
  { $match: { year: 2014 } },
  { $redact:
    { $cond:
      { if:
        {$gt: [{$size: {$setIntersection: ["$tags", userAccess]}}, 0] },
        then: "$$DESCEND",
        else: "$$PRUNE"
      }
    }
  }
] )
```

The result of executing this query given the sample document from above would be:

```
{
  "_id" : 1,
  "title" : "123 Department Report",
  "tags" : [ "low" ],
  "year" : 2014,
  "subsections" : [
    {
      "subtitle" : "Section 1: Overview",
      "tags" : [ "low" ],
      "content" : "Section 1: This is the content of section 1."
    }
  ]
}
```

The sections tagged with the "`medium`" and "`high`" security markings were redacted from the document since the user's access level was only "`low`".

CAPCO Document Markings

A common security marking scheme within the US Government is commonly referred to as CAPCO. CAPCO markings include multiple security markings such as classification, SCI controls, and dissemination controls. The CAPCO security model essentially requires that a user has a superset of the accesses that are marked in a given document or portion within a document (see http://www.ncix.gov/training/resources/Publically%20Releasable%20Register.pdf for reference).

For example, a “TOP SECRET//SI/TK//NOFORN” marking means that a person must have the “TOP SECRET” clearance AND the “SI” *AND* the “TK” SCI compartments *AND* not be a foreign national. This requires our marking scheme to support the AND boolean operator.

However, CAPCO dissemination controls (eg “REL TO”) can also specify a list countries or groups that person must be a citizen/member of one or more of in order to see the document or portion of the document.

For example, a “TOP SECRET//SI/TK//REL TO USA, GBR” marking means that a person must have the “TOP SECRET” clearance AND the “SI” *AND* the “TK” SCI compartments *AND* be a citizen of USA *OR* GBR. This requires our marking scheme to also support the OR boolean operator.

To represent this in JSON, an array of nested arrays is used. The outer arrays are for AND and the the inner arrays are for OR.

For example “[[A], [B], [C, D]]” means A *AND* B *AND* (C *OR* D). To make this into valid JSON, field names are added to set what each control is. For CAPCO, the following field names are used (this is not a complete mapping and is only here for illustrative purposes):

CAPCO Control	**Field**
Classification	c
SCI	sci
Authorized for Release to	relto

So to represent the “TOP SECRET//SI/TK//REL TO USA, GBR” marking, the following JSON will be used:

```
[ [{"c" : "TS"}], [{"sci" : "SI"}], [{"sci" : "TK"}], [{"relto" :
"USA"}, {"relto" : "GBR"}] ]
```

CAPCO Document Markings Example

The document below shows a sample document with subsections containing CAPCO security markings. In this example, the "sl" (stands for “security label”) field, which is an array that contains one or more arrays of objects as described above, is used to contain the marking. The “sl” field name was chosen for compactness but this could be any valid field name that make sense for a particular schema or marking scheme.

```
{
  _id: 1,
  title: "123 Department Report",
  year: 2014,
  subsections: [
    {
      subtitle: "Section 1: Overview",
      sl: [ [{"c" : "U"}] ],
      content:  "Section 1 Content..."
    },
    {
      subtitle: "Section 2: Analysis",
      sl: [ [{"c" : "S"}], [{"sci" : "SI"}] ],
      content: "Section 2 Content..."
    },
    {
      subtitle: "Section 3: Budgeting",
      sl: [ [{"c" : "TS"}], [{"sci" : "SI"}], [{"sci" : "TK"}] ],
      content: "Section 3 Content..."
    }
  ]
}
```

Each of the objects in the “sl” arrays are a single key/value pair (e.g. `{"c" : "U"}` which represents the classification).

CAPCO Redaction Expression

Since the CAPCO security model requires that the user has a superset of the accesses that are marked in a given document, a redaction expression that implements that logic based on a specific security marking scheme needs to be created.

For example, if the following security marking is used:

```
sl: [ [{"c" : "S"}], [{"sci" : "SI"}], [{"sci" : "TK"}] ],
```

In order for a user to access that document or sub-document, the user would need to have a classification of "S" or higher (classifications are hierarchical) **and** the user would need to have SCI controls "SI" **and** "TK", because each is in a separate sub-array.

Implementing this logic requires a slightly more complex redaction expression and document marking scheme than the previous redaction expression example.

CAPCO Redaction Expression Example

In the `mongo` shell example below, an example set of user accesses is declared, a CAPCO redaction expression is constructed and applied to an aggregation pipeline using the $redact pipeline stage. In this example the highest classification access for the user is "TS", and since classification is hierarchical, the list of user access for classification is expanded to include the lower classification levels "S" and "U".

```
var userAccess = [{"c":"U"}, {"c":"S"}, {"c":"TS"}, {"sci":"SI"}];
var capcoRedactExpr = {
  $cond: {
    if: {
      $allElementsTrue : {
        $map : {
          input: { $ifNull:[ "$sl",[[]]] },
          "as" : "setNeeded",
          "in" : { $cond: {
            if: {
              $or: [
                { $eq: [ { $size: "$$setNeeded" }, 0 ] },
                { $gt: [
                  {
                    $size: {
                      $setIntersection: [ "$$setNeeded", userAccess ] }
                  },
                  0 ]
                }
              ]
            },
            then: true,
            else: false
          } }
        }
      }
    },
    then: "$$DESCEND",
    else: "$$PRUNE"
  }
}
```

```
db.report.aggregate( [
  { $redact: capcoRedactExpr },
  { $match: { year: 2014 } }
] )
```

Given our example CAPCO document above, here is the result of executing the redaction pipeline:

```
{
  _id: 1,
  title: "123 Department Report",
  year: 2014,
  subsections: [
    {
      subtitle: "Section 1: Overview",
      sl: [ [{"c" : "U"}] ],
      content:  "Section 1 Content..."
    },
    {
      subtitle: "Section 2: Analysis",
      sl: [ [{"c" : "S"}], [{"sci" : "SI"}] ],
      content: "Section 2 Content..."
    }
  ]
}
```

The Section 3 subsection was redacted from the document because it has SCI controls "SI" and "TK" but the user only has the "SI" SCI control access.

Example Implementation

The Field-Level Access Control (FLAC) example implementation is a set of example Java classes that illustrate implementing MongoDB Field-Level Redaction.

There are 4 major components required to make FLAC work:

1. Documents marked with a specific set of security markings
2. A class used to represent the security attributes for a given user
3. A redaction expression implementing your domain specific security model (e.g. CAPCO)
4. A class that will apply the user's security attributes to a redaction expression, constructing and executing an aggregation pipeline with the specified query criteria and options.

Document/sub-document Markings

As shown in the CAPCO examples above, documents are annotated by marking the fields that you need to protect with a security marking. The FLAC example code uses the “sl” field for compactness but any valid field name can be used.

User Attributes

The FLAC implementation base classes is designed to support different marking schemes and redaction logic. The example code uses a generic class called SecurityAttributes which encapsulates user attributes that are used in the redaction logic.

The `CapcoSecurityAttributes` class is a subclass of `SecurityAttributes` that encapsulates user attributes that are relevant to CAPCO visibility logic such as clearance level, SCI compartments and citizenship. It also handles the logic of mapping those attributes to the values that match the marking scheme used in the documents (IE, “clearance” maps to “c”).

Each query to the database would be done in the context of a particular user. A `SecurityAttributes` instance would be created to capture the user’s relevant attributes and passed to the code executing the query.

Redaction Expression

The `RedactExpression` interface defines a single method that is used to retrieve a JSON string containing the redaction expression that should be used for a particular marking scheme. This string can then be parsed into a `DBObject` and used as a stage in an aggregation pipeline. The `StringRedactExpression` class provides the basic implementation for substituting the security label field name and the users attributes into the a redaction expression specified via the constructor.

The `CapcoRedactExpression` contains the actual redaction expression (described earlier) used to implement the “AND of ORs” logic required for CAPCO.

DBCollection Wrapper

Access to MongoDB via the Java driver is done through the `DBCollection` class. The FLAC example implementation provides a wrapper class called `RedactedDBCollection` that is used to interface with mongodb in a FLAC-aware manner. `RedactedDBCollection` offers the usual `find()` and `aggregate()` methods to execute queries. It wraps each query in an aggregation pipeline that applies the given redaction expression and user attributes to implement the desired redaction behavior.

Appendix - Working Example

The `com.mongodb.flac.RedactedDBCollectionTest.java` source file has some examples of how to use the `RedactedDBCollection` class. Here we'll walk through a simple program that runs a query against a FLAC-controlled collection.

Step by Step

First we choose to setup the query to match documents with "firstName" equal to "Alice" and we set the projection to an empty object so we get all fields back

```
BasicDBObject query = new BasicDBObject("firstName", "Alice");
BasicDBObject keys = new BasicDBObject();
```

Next, we connect to the database and get a `DBCollection`, this is not protected by FLAC

```
Mongo mongo = new Mongo();
DB db = mongo.getDB("test");
DBCollection srcCollection = db.getCollection("test");
```

Next, we construct the object needed to specify the current user's security attributes and the redaction expression, specifying the security marking field name , "sl", used within the documents

```
CapcoSecurityAttributes userAttributes = new CapcoSecurityAttributes();
userAttributes.setClearance("TS");
userAttributes.setSci(Arrays.asList("TK"));
CapcoRedactExpression capcoRedactExpression = new CapcoRedactExpression("sl");
```

Next, we construct the protected `RedactedDBCollection` that wraps the native, unprotected DBCollection

```
RedactedDBCollection redactedCollection = new
RedactedDBCollection(srcCollection, userAttributes, capcoRedactExpression);
```

Finally, we run the query and process the results

```
Cursor results = redactedCollection.find(query, keys);
while (results.hasNext()) {
  DBObject doc = results.next();
  // do something with the DBObject
```

```
}
```