

GradeProof API & Dashboard

COMP3615 Major Project Documentation

GradeProof Team



Contents

1	Introduction	5
1.1	Aim	5
1.2	About the API & Dashboard	5
2	Deployment	7
2.1	Setting Up Elasticsearch	7
2.2	Updating Elasticsearch	8
2.3	Setting Up API & Dashboard	11
3	The Product	13
3.1	The API	13
3.1.1	Routes	13
3.1.2	API Security	13
3.2	The Dashboard	14
3.2.1	Users Page	14
3.2.2	Documents Page	15
3.2.3	Platforms Page	16
3.2.4	Rules Page	17
4	Elasticsearch walkthrough	19
4.1	Basic Example	19
4.2	More Complicated Example	21

A close-up photograph of a person's hand holding a black fountain pen, poised to write on a white document. The hand is wearing a dark suit jacket and a white shirt cuff. The background is blurred, showing a wooden desk. A red rounded rectangular box is overlaid on the image, containing the section header.

1. Introduction

1.1 Aim

Beginning from August, the COMP3615 Gradeproof team has come together to build an API and dashboard for our client. The aim of the project was to build a tool to interactively access data about how our client's consumers interacted with the GradeProof service. This documentation will illustrate the tool we have created for our client, including technical details about how our product fits in with tools that our product is interacting with.

1.2 About the API & Dashboard

The API was designed for the initial purpose of accessing informative data and allowing functionality of the API to be used across other services. Through the API, you will be able to directly query Elasticsearch for data and see how the users are interacting with the GradeProof service.

The Dashboard is built into the API and provides a user-friendly visual with multiple options to edit and adjust query searches. From the dashboard, you will be able to query from user, detection, document, dictionary and platform tables. Without doubt, the dashboard provided is more adaptable, customisable and intuitive than Kibana.

A close-up photograph of a person's hand holding a black fountain pen, poised to write on a white document. The pen is held in a tripod grip. The document is on a desk with a red leather binding. The background is dark and out of focus.

2. Deployment

This chapter of the report covers the setup and maintenance to run the Gradeproof API and dashboard. Note: The deployment chapter is modeled from the test data provided during the development of the API & dashboard.

2.1 Setting Up Elasticsearch

The Flask API was setup for using a data-hierarchy structure in Elasticsearch. In order to replicate this Elasticsearch data for production data, first create the production elasticsearch DB on amazon AWS, then follow the following instructions:



- go to directory `/gradeproof_api`
- type `/manage.py setup_elasticsearch`
- follow the instructions on screen
- you will now have a `.env` file in that directory which you may edit at any time.

If you do not follow these instructions, the app will not work.

2.2 Updating Elasticsearch

Relatively low maintenance is required for Elasticsearch however in order to keep Elasticsearch updated with the constant in take of data from DyanmoDb, we require some Lambda scripts to be implemented.

Configure triggers
Configure an optional trigger to automatically invoke your function.

DynamoDB 
▶ 
Lambda Remove

DynamoDB table ⓘ

Batch size ⓘ

Starting position ⓘ

In order to read from the DynamoDB stream, your execution role must have proper permissions.

Enable trigger ⓘ

Cancel
Previous
Next

Figure 2.1: In blank function, set trigger to Dynamodb

Once you have created a new Lambda function and enabled the trigger between DynamoDB and the new function, you must configure the code:

1. Within the GradeProof repository go to *AWS-Lambda-code/FinalLambdaCopy/*
2. Within the directory is four scripts named after the corresponding table in Dynamodb, and a requests.zip file
3. We must now create an "AWS Deployment Package", this is necessary to add the python Requests module to the Lambda function.
4. Unzip the requests.zip file to extract the python module, Requests.
5. Select the newly unzipped Requests folder, and the appropriate python file for the table you are triggering, and compress them into a .zip folder. Note: The files must be in the root of the .zip folder, NOT inside a folder
6. Return to the AWS Lambda function page, and go to the "Code" tab.
7. Locate the Dropdown box titled "Code entry type". Change this to "Upload a .zip file", and upload the newly created .zip file.
8. Once the file has uploaded, you may now change the "Code entry type" back to "Edit code inline" to make any modifications.
9. You will have to modify the URL pointing to the production Elasticsearch Instance. Additionally, you will have to modify the "_index" in the "Header" dictionary. Change this to your desired Index, but remember to keep this consistent amongst the other files.

10. Finally, on the AWS Lambda page, go to the "Configuration" tab, and modify the "Handler" field to `nameOfPythonFileWithoutFileExtension._lambda_handler`
11. Repeat this process for the other 3 remaining tables, and their corresponding Lambda functions.

Configure function

A Lambda function consists of the custom code you want to execute. [Learn more](#) about Lambda functions.

Name*

Description

Runtime*

Lambda function code

Provide the code for your function. Use the editor if your code does not require custom libraries (other than boto3). If you need custom libraries, you can upload your code and libraries as a .ZIP file.

Code entry type

```
16
17- bool(the_string):
18- "false" in the_string:
19-     return False
20-     return True
21
22
23- lambda_handler(event, context):
24-     temp = event['Records']
25-     hack_the_system = json.dumps(temp)
26-     records = json.loads(hack_the_system)
27-     load = ""
28
29-     for record in records:
30-         r = record.get("dynamodb").get("NewImage")
31-         detection = {}
32-         detection["addedToDictionary"] = to_bool(r["addedToDictionary"].get("S"))
33-         detection["category"] = r["category"].get("S")
34-         detection["document"] = r["document"].get("S")
35-         detection["context"] = r["context"].get("S")
36-         detection["detectedAt"] = int(r["detectedAt"].get("N"))
37-         detection["id"] = r["id"].get("S")
38-         detection["ignored"] = to_bool(r["ignored"].get("S"))
39-         detection["ignoredAll"] = to_bool(r["ignoredAll"].get("S"))
```

Figure 2.2: Fillout the details, and Select upload zip under code entry to add the code

Lambda function handler and role

Handler*

Role* ⓘ

Existing role* ⓘ

Figure 2.3: Set Role: "Choosing an existing role" & Existing role: "ddb-elasticsearch-bridge"

Review

Please review your Lambda function details. You can go back to edit changes for each section. When you are ready, click **Create function** to complete the setup process.

Triggers

[Edit](#)

DynamoDB

DynamoDB table: **DEV_Document** Batch size: **100** Starting position: **LATEST**

Enabled

Lambda function

[Edit](#)**Name** update_detections**Description** Updates detections to ElasticSearch**Runtime** Python 2.7**Handler** lambda_function.lambda_handler**Existing role*** ddb-elasticsearch-bridge**Memory (MB)** 128**Timeout** 3**VPC** No VPC[Cancel](#)[Previous](#)[Create function](#)

Figure 2.4: Review what you have done and click Create function

2.3 Setting Up API & Dashboard

This is the initial setup used for local machine:

- Make sure you have Python3 installed which will include the package manager, pip.
- Open a terminal and change directory into the `gradeproof_api` folder within the root directory
- Install virtualenv through pip. Using a virtual environment will reduce the cluttering of installing global packages. ***pip install virtualenv***
- Create your virtual environment ***virtualenv -p python3 venv***
- Start virtual environment ***source venv/bin/activate***
- Install all packages required for the application: ***pip install -r requirements.txt*** This will install your packages within your virtual machine.
- After the packages are installed, run application: ***python app.py*** in the directory containing `app.py`
- Open a browser and in the address bar type ***http://localhost:5000*** and press enter.
- You have now successfully installed and run the GradeProof statistics dashboard.



3. The Product

3.1 The API

3.1.1 Routes

The following routes are available

<code>api_dictionary.get_dictionary_data</code>	GET	<code>/api/dictionary/</code>
<code>api_platforms.platforms_accepted_rate</code>	GET	<code>/api/platforms/</code>
<code>query.match_query</code>	GET	<code>/api/query/match</code>
<code>api_rules.get_rules</code>	GET	<code>/api/rules/</code>
<code>api_rules.get_rule</code>	GET	<code>/api/rules/[rule_id]/context</code>
<code>api_rules.most_common_used_rules</code>	GET	<code>/api/rules/common</code>
<code>api_users.get_user</code>	GET	<code>/api/users/[user_id]</code>
<code>api_users.single_user_aggregate</code>	GET	<code>/api/users/[user_identifier]/aggregate</code>
<code>api_users.all_user_aggregate</code>	GET	<code>/api/users/aggregate</code>
<code>api_users.find_user</code>	GET	<code>/api/users/search</code>

The first column indicates which python module and method is assigned to which api route.

3.1.2 API Security

Unfortunately at the time of this writing, the security of the API itself were not finished in time. However, we did implement best practices in keeping sensitive information out of the source code with the use of `.env` files as explained earlier.

In the file `/gradeproof_api/client.py` you will see a stub to add a secure check onto elasticsearch using amazon to ensure that only your service can access it. This should be enough to cover elasticsearch itself, though the API itself will still need securing.

3.2 The Dashboard

The dashboard is created using Vue.js - a frontend UI library. The unfortunate side of this is to make changes to the frontend it must be recompiled. To do so, you must have node/npm installed, and after having the required packages installed (by navigating the main directory and typing `npm install`):

Compiling

- navigate to `/main`
- type `node_modules/gulp/bin/gulp.js`

3.2.1 Users Page

Description

The users page is used for all our user related queries using the User table. On most occasions data is aggregated when requesting more data from other tables.

The users page provides the following information:

- User's details
- Dictionary words added from User
- Platforms that the user has interacted with

Functionality

The users page provides the following functionality:

- Tables that provide user usage over time data.
- A search function that finds a users aggregated data.

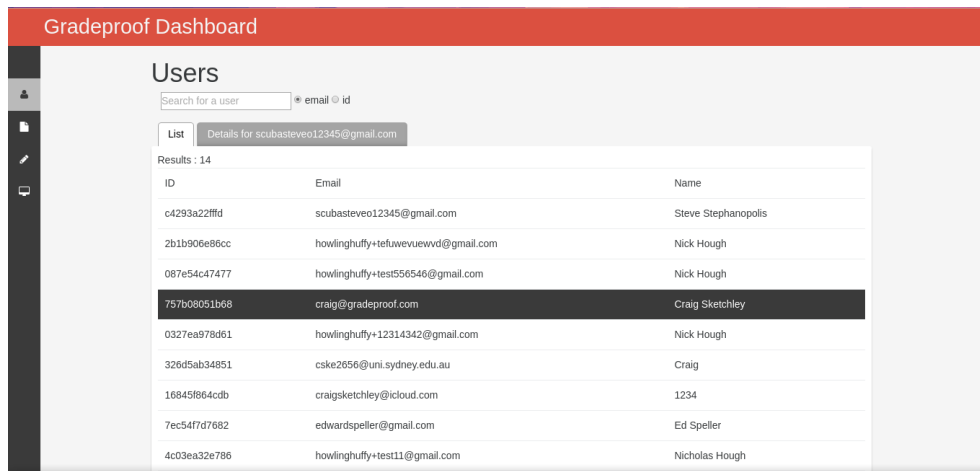


Figure 3.1: Search for users and navigate to user page by clicking user

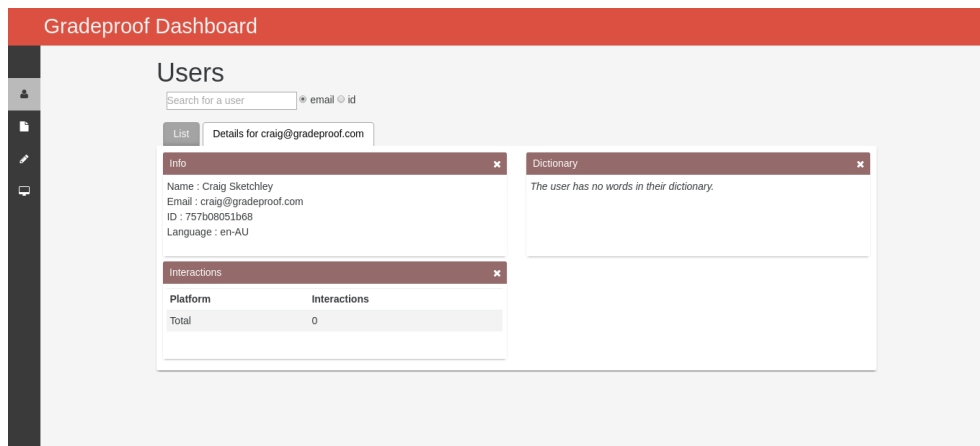


Figure 3.2: Find user details

3.2.2 Documents Page

The documents page is used platform related queries.

The documents page provides the following information:

- Shows the counts for query based search results. i.e as a result

Functionality

The platforms page provides the following functionality:

- Search function to query text

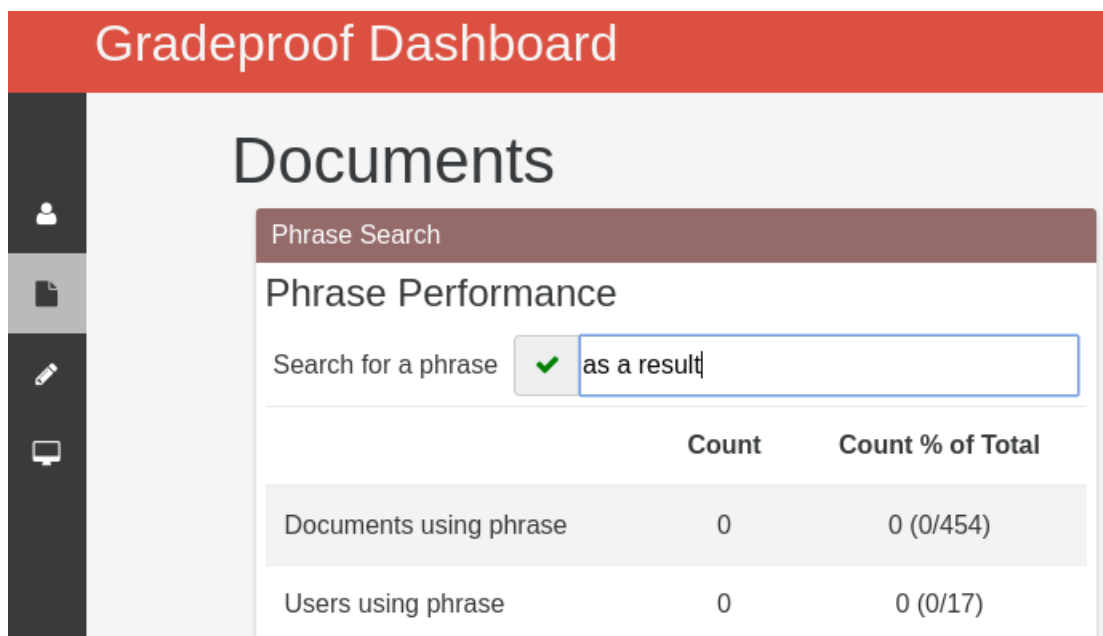


Figure 3.3: Counts from query search

3.2.3 Platforms Page

The platforms page is used platform related queries.

The platforms page provides the following information:

- Shows the count per platform displayed in a chart

Functionality

The platforms page provides the following functionality:

- No, current customisation added as of yet.

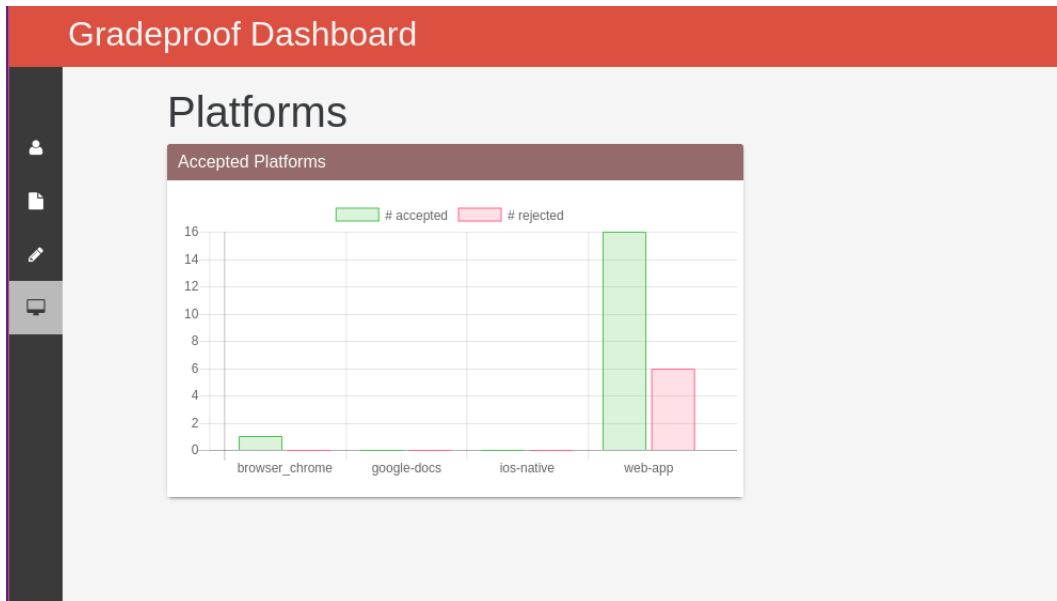


Figure 3.4: Counts per platforms displayed here

3.2.4 Rules Page

The rules page is used for all our rule related queries using the detection table.

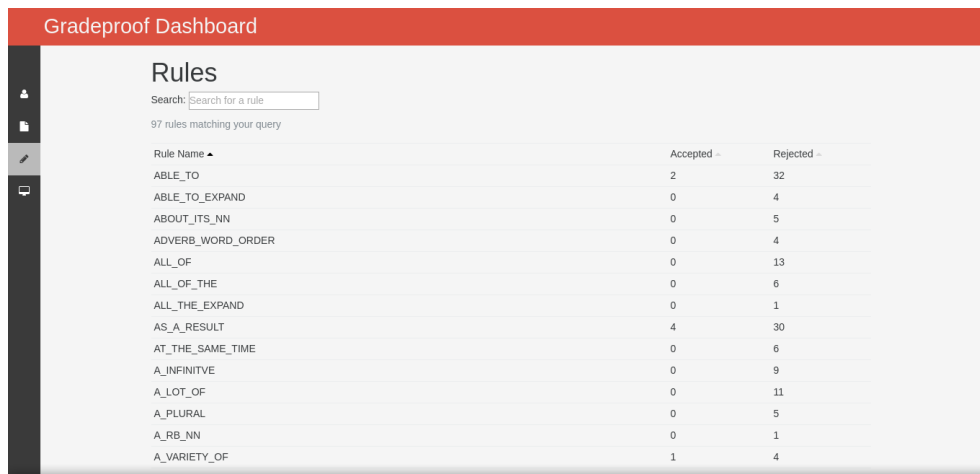
The rules page provides the following information:

- The acceptance and rejection rate for the Rules displayed in a table.
- Text context where the rule is accepted and rejected.
- Rule information about the text case

Functionality

The rule page provides the following functionality:

- Text context where a rule has been breached.
- A search function that finds a particular rule.



Gradeproof Dashboard

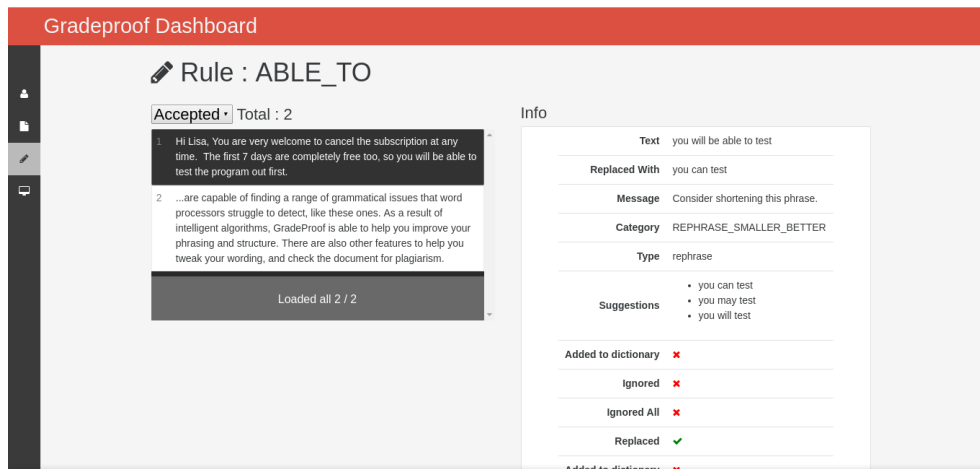
Rules

Search:

97 rules matching your query

Rule Name	Accepted	Rejected
ABLE_TO	2	32
ABLE_TO_EXPAND	0	4
ABOUT_ITS_NN	0	5
ADVERB_WORD_ORDER	0	4
ALL_OF	0	13
ALL_OF_THE	0	6
ALL_THE_EXPAND	0	1
AS_A_RESULT	4	30
AT_THE_SAME_TIME	0	6
A_INFINITIVE	0	9
A_LOT_OF	0	11
A_PLURAL	0	5
A_RB_NN	0	1
A_VARIETY_OF	1	4

Figure 3.5: Search for rules and navigate to rule page by clicking rule



Gradeproof Dashboard

Rule : ABLE_TO

Accepted Total : 2

- Hi Lisa, You are very welcome to cancel the subscription at any time. The first 7 days are completely free too, so you will be able to test the program out first.
- ...are capable of finding a range of grammatical issues that word processors struggle to detect, like these ones. As a result of intelligent algorithms, GradeProof is able to help you improve your phrasing and structure. There are also other features to help you tweak your wording, and check the document for plagiarism.

Loaded all 2 / 2

Info

Text	you will be able to test
Replaced With	you can test
Message	Consider shortening this phrase.
Category	REPHRASE_SMALLER_BETTER
Type	rephrase
Suggestions	<ul style="list-style-type: none"> • you can test • you may test • you will test
Added to dictionary	✗
Ignored	✗
Ignored All	✗
Replaced	✓
Added to dictionary	✗

Figure 3.6: Select Accepted or Rejected to see the context of the rule



4. Elasticsearch walkthrough

4.1 Basic Example

In this section we will walkthrough a basic Elasticsearch query explaining what each level of it does. The purpose of this query is to return the number of interactions with a specific rule. This query would be searching the 'detection' table. Here is the entire query:

```
{
  "size": 0,
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            { "term": { "ignored": true }},
            { "term": { "ignoredAll": true }},
            { "term": { "removed": true }},
            { "term": { "addedToDictionary": true }},
            { "exists": { "field": "replacedWith" }}
          ],
          "must": [
            { "term": { "languageToolRuleId": "morfologik_rule_en_au" }}
          ]
        }
      }
    }
  }
}
```

Now let's go through the query line by line.

```
"size": 0,
```

The first line is the size parameter. It tells our query how many results to return. In this case 0 as we do not want to actually see any of the detections we just want a count of them.

```
"query": {
```

The query parameter, everything following this is the body of our query.

```
"constant_score": {
```

Usually when returning results Elasticsearch scores each result on how well it matches the query. As we just want to include/exclude documents we use the constant score parameter and the results are given a uniform score of one (even though we aren't returning any results).

```
"filter": {
```

The filter parameter puts the query in *filter* context, meaning no scores will be calculated for results, the return will be a simple yes or no. It can be used as a parameter in the constant score query as here.

```
"bool": {
```

The bool query is placed inside a constant score query. It allows us to build queries with multiple clauses.

```
"should": [
```

Should is the same as saying at least one of these clauses is must match for a document to be returned as an interaction. It is the equivalent of an OR statement.

```
{ "term": { "ignored": true }},
{ "term": { "ignoredAll": true }},
{ "term": { "removed": true }},
{ "term": { "addedToDictionary": true }},
```

Term queries find documents that contain the exact term specified in the inverted index of a specified field.

```
{ "exists": { "field": "replacedWith" }}
```

Exists queries return documents that have at least one non-null value in the original field.

```
"must": [
  { "term": { "languageToolRuleId": "morfologik_rule_en_au" }}
```

As 'should' is to OR 'must' is to AND. For a document to be returned it must match every clause specified under a must.

4.2 More Complicated Example

Our more complicated example will search for the number of interactions again. This time for a specific user id and in a specific date range. This example is again searching the *detection* table. Full query:

```
{
  "size": 0,
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            { "term": { "ignored": true }},
            { "term": { "ignoredAll": true }},
            { "term": { "removed": true }},
            { "term": { "addedToDictionary": true }},
            { "exists": { "field": "replacedWith" }}
          ],
          "must": [
            { "has_parent": {
              "type": "document",
              "query": {
                "has_parent": {
                  "type": "user",
                  "query": {
                    "match": {
                      "_id": "7ec54f7d7682"
                    }
                  }
                }
              }
            }
          ]
        }
      },
      { "range": {
        "detectedAt": {
          "gte": 1400000000000,
          "lte": 1500000000000
        }
      }
    }
  }
}
```

Now let's go through it one section at a time again.

```
{
  "size": 0,
  "query": {
    "constant_score": {
      "filter": {
        "bool": {
          "should": [
            { "term": { "ignored": true }},
            { "term": { "ignoredAll": true }},
            { "term": { "removed": true }},
            { "term": { "addedToDictionary": true }},
            { "exists": { "field": "replacedWith" }}
          ]
        }
      }
    }
  }
}
```

This section is exactly the same as the last query. We are doing a yes/no selection of only those detections that have been interacted with.

```
"must": [
  { "has_parent": {
    "type": "document",
```

This is where our query starts to differ. Under the must section of the bool query we are running a has parent query. The has parent query will return children based on data in their parents, in this case detections based on the data of their parent documents which is specified in the type parameter.

```
"query": {
  "has_parent": {
    "type": "user",
```

Seeing as we are looking for a user id, we need to use another has parent query on the document select based on data in its parent, user.

```
"query": {
  "match": {
    "_id": "7ec54f7d7682"
  }
}
```

Our final query within these nested parent-child queries is a match query based on the user id.

```
{ "range": {
  "detectedAt": {
    "gte": 1400000000000,
    "lte": 1500000000000
  }
}}
```

This is the last section of our query. Note that it is outside of the has parent queries from before but still under the must section of the bool query. This means it is a query on the detection field that must be satisfied. It is also a range query, which as the name suggests allows us to search for results in a range. 'detectedAt' tells the query which field of the data to search. 'gte' means greater than and 'lte' means less than. The times here are Unix time stamps.

That concludes our brief walkthrough of two Elasticsearch queries. There is a huge amount more functionality to Elasticsearch. The Elasticsearch guide is the best place to start if you want to learn Elasticsearch quickly . Then the Elasticsearch reference is great if you want to learn more about specific functionality.

- [Guide](#)
- [Reference](#)