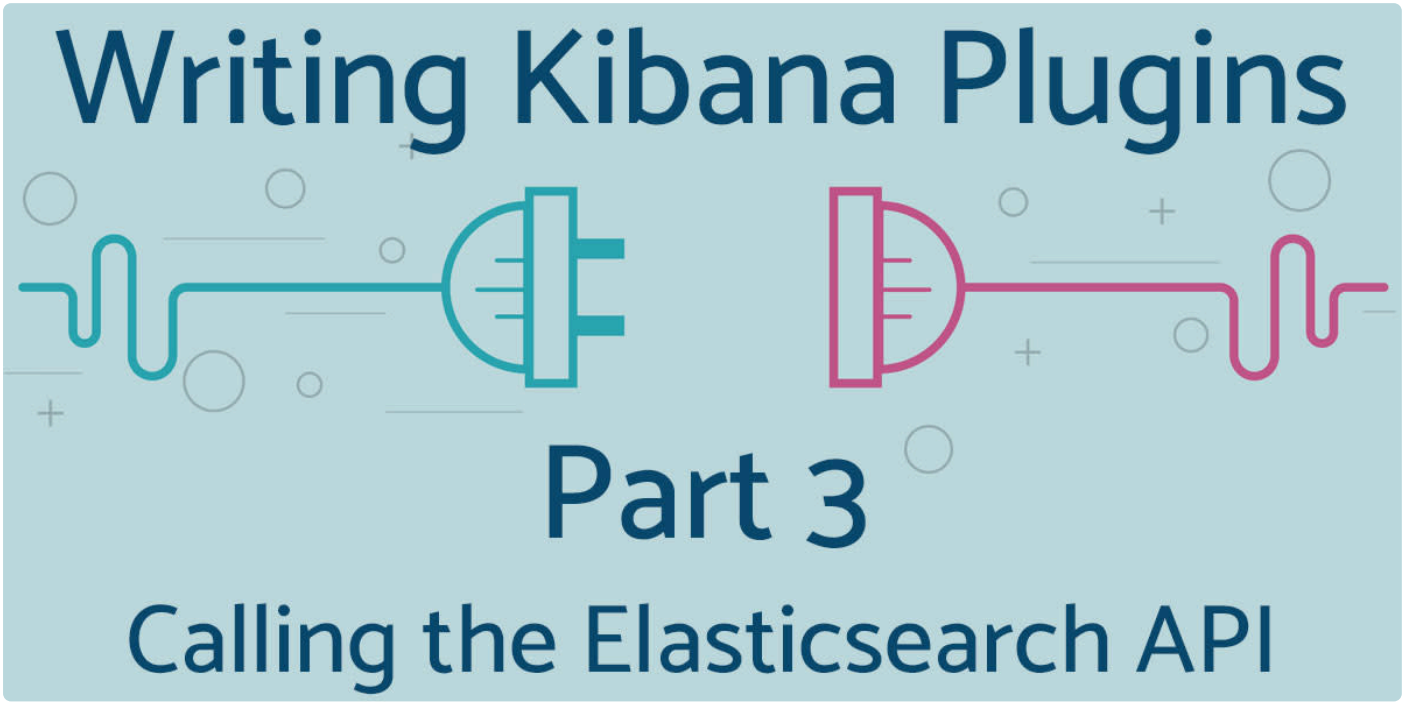
Writing Kibana Plugins - Part 3: Calling the Elasticsearch API

Jochen Kressin · 2023-03-10



This article explores how a Kibana plugin can communicate with Elasticsearch to index and fetch data. Have fun!

The completed plugin with all modifications we do in this article can be found under the following link https://git.floragunn.com/search-guard/sg*kibana*demo_plugin

Our Demo Use Case

We want to manage account data in Elasticsearch and provide a friendly Kibana UI to manage them.

The UI should have the following capabilities:

- List accounts in a table
- Search for accounts
- Clone an account
- Delete an account

To accomplish this we can split it into the subtasks:

- Index the data if it doesn't exist.
- Add a server API to fetch and update the data.
- Add a UI to manage the data.

Server - Indexing Sample Data

We use randomly generated sample account data from https://www.json-generator.com/. For our demo we do not need the full set of available fields, so we only select a couple of them. A generated account might look like this:

```
[
  {
    "_id": "640e1eaacbe89bae2eb62217",
    "name": "Monique Pennington",
    "balance": "$3,996.80",
    "gender": "female",
    "company": "VINCH",
    "phone": "+1 (843) 565-3143",
    "address": "247 Bokee Court, Fulford, Tennessee, 6878",
    "registered": "2021-09-12T12:55:16 -02:00",
    "friends": [
      {
        "id": 0,
        "name": "Tracy Woodard"
      },
      {
        "id": 1,
        "name": "Katherine Barron"
      },
      {
        "id": 2,
        "name": "Sophia Vasquez"
      }
    ]
  },
  ...
]
```

Create the data and put it under **server/dataset/searchguard*demo*accounts.json**

```
sg_kibana_demo_plugin/
├── server
│   ├── dataset
│   │   └── searchguard_demo_accounts.json
```

We want to index this sample data if the index does not already exist on our cluster. Before starting the indexing process, we must check the Elasticsearch cluster status. We index the data only if the status is green or yellow. If the status is red, we retry the status call every 3 seconds until it is not red.

First, let's define the name of the index we want to store the data as a constant:

common/index.ts

```
export const INDEX_NAME = 'searchguard_demo_accounts';
```

Then we add two functions in server/plugins.ts which carry out the actual indexing operations.

server/plugins.ts

```
import { INDEX_NAME } from '../common';
import datasetJSON from './dataset/searchguard_demo_accounts.json';

function indexDatasetOnceIfAbsent({ clusterClient, logger }) {
  return async function () {
    const { body: doesExist } = await clusterClient.asInternalUser.indices.exists({
      index: INDEX_NAME,
    });

    if (!doesExist) {
      logger.info(
        `Index ${INDEX_NAME} doesn't exist. Start indexing data, it will take some time, please
wait ...`
      );

      try {
        await clusterClient.asInternalUser.indices.create({ index: INDEX_NAME });
      } catch (error) {
        throw new Error(`Failed to create index ${INDEX_NAME}. ${error.stack}`);
      }

      // Create dataset that matches requirements of the
      // Elasticsearch Bulk API payload.
      const dataset = datasetJSON.flatMap(({ _id, ...doc }) => [
        { index: { _id, _index: INDEX_NAME } },
        doc,
      ]);

      let bulkResp;

      try {
        bulkResp = await clusterClient.asInternalUser.bulk({ refresh: true, body: dataset });
        bulkResp = bulkResp.body;
        logger.info(`Finished indexing data in ${INDEX_NAME} index!`);
      } catch (error) {
        throw new Error(`Failed to index the dataset. ${error.stack}`);
      }

      // Log errors if any.
      if (bulkResp && bulkResp.errors) {
        const errored = [];
        bulkResp.items.forEach((action, i) => {
          const operation = Object.keys(action)[0];
          if (action[operation].error) {
            errored.push({
              status: action[operation].status,
              error: action[operation].error,
              operation: dataset[i * 2],
              document: dataset[i * 2 + 1],
            });
          }
        });

        logger.error('Some documents were not indexed. Probably a mapping issue.');
        logger.error(errored);
      }
    }
```

```
  };
}

function retryClusterCall({ clusterClient, logger, retryInMS = 3000 }, fn) {
  let interval;
  interval = setInterval(async () => {
    try {
      const { body: { status = 'red' } = {} } = await
clusterClient.asInternalUser.cluster.health();
      logger.info(`Elasticsearch cluster health is ${status}.`);

      if (status === 'green' || status === 'yellow') {
        clearInterval(interval);
        if (typeof fn === 'function') {
          fn();
        }
      }
    } catch (error) {
      logger.error(error);
    }
  }, retryInMS);
}
```

The first function (*indexDatasetOnceIfAbsent*) checks if our target index already exists by calling the Elasticsearch *indices.exists* API. If this is not the case, the index is created and the sample data is uploaded by using the Elasticsearch *bulk* API. Before that, we transform the sample data JSON in a JSON format expected by the *bulk* API.

If the bulk upload does not succeed, we loop over the items in the error message, extract the erroneous ones and construct and log a meaningful message.

The Nitty-gritty Details

Requests to Elasticsearch

Let's have a look at some details and some recurring patterns. The first thing you might notice is the use of *clusterClient*. This is a reference to the Node.js client for Elasticsearch. It is the most common way to talk to Elasticsearch from Kibana, and Kibana provides instances of this client for plugins to use. The client has an API that exposes the most common Elasicsearch features like index, bulk, search, scroll etc. Elastic publishes a complete API documentation for your reference here.

You will also notice that instead of the direct calls to the client like

```
client.indices.exists
```

we use a call like

```
clusterClient.asInternalUser.indices.exists
```

This has to do with authentication and authorization. If Elasticsearch is protected by a security plugin like Search Guard or X-Pack security, we need to tell Kibana which user is actually issuing the API call. There

are two options: Make the call "as the Kibana server" or in the context of the currently logged-in user. We will not go into user and identity management details in Kibana, so we make all calls as the Kibana server user (sometimes called the Kibana system user).

Async Calls

All of the calls to Elasticsearch are *async* calls. Thus, you can either work with *Promises* or, as in our example, use the *await* keyword to wait for the async call to finish. Note that when using await, this call will be a blocking call.

```
try {
  await clusterClient.asInternalUser.indices.create({ index: INDEX_NAME });
} catch (error) {
  throw new Error(`Failed to create index ${INDEX_NAME}. ${error.stack}`);
}
```

The call above uses the cluserClient to create an index. Because we use await, the call blocks until the index is created successfully or an error is thrown. We catch the error and throw a more meaningful error to the caller.

Logging

Instead of using the JavaScript built-in logging capabilities like *console.warn()*, *console.error()* etc. you should use the logging service provided by Kibana. The logger object has all methods you already know from *console*, like *logger.warn()* or *logger.error()*.

The nice thing about the logging service is that it provides a logging context for each logger. You can set the context when obtaining the logger object from the plugin initializer like:

```
logger = initializerContext.logger.get('searchguard', 'sampledata');
```

You can now log a message on error level like:

```
logger.error('something went wrong')
```

This will show up in the Kibana log as follows:

```
[2023-03-02T10:01:23.452-08:00][ERROR][searchguard.sampledata] something went wrong
```

Note: The logging service is only available on the server side

Calling Our Code on Plugin Startup

We can now call our methods on plugin startup. For this, we extend the server plugin *start()* method discussed in the last article.

```
export class SgKibanaDemoPluginPlugin
  implements Plugin<SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart> {

  public start(core: CoreStart) {
    this.logger.debug('sgKibanaDemoPlugin: Started');

    // Index datase only if Elasticsearch cluster is ready
    retryClusterCall(
      {
        logger: this.logger,
        clusterClient: core.elasticsearch.client,
      },
      indexDatasetOnceIfAbsent({
        logger: this.logger,
        clusterClient: core.elasticsearch.client,
      })
    );

    return {};
  }
}
```

After Kibana starts up, you should see the newly created index on Elasticsearch:

```
$ curl -X GET http://localhost:9200/_cat/indices
yellow open searchguard_demo_accounts  Jz5GEh1YRgOy3JLAOWeAJA 1 1  997  0   1.4mb   1.4mb
```

Next Steps

In the following article, we will add some server-side routes to our plugin, making it possible to get, add and delete accounts.

In the last article, we will turn to the client side and add a friendly React-based UI to the plugin. Stay tuned!

Where to go Next

- Elasticsearch JavaScript client API
- Kibana logging service
- Writing Kibana Plugins, part 1: Plugin Basics
- Writing Kibana Plugins, part 2: A Skeleton Plugin
- Writing Kibana Plugins, part 4: Adding Routes
- Writing Kibana Plugins, part 5: Adding a User Interface (coming soon)
- If you have any questions, don't hesitate to ask on our forum

This article was published on the Search Guard blog: https://search-guard.com/blog/writing-kibana-plugins-3/