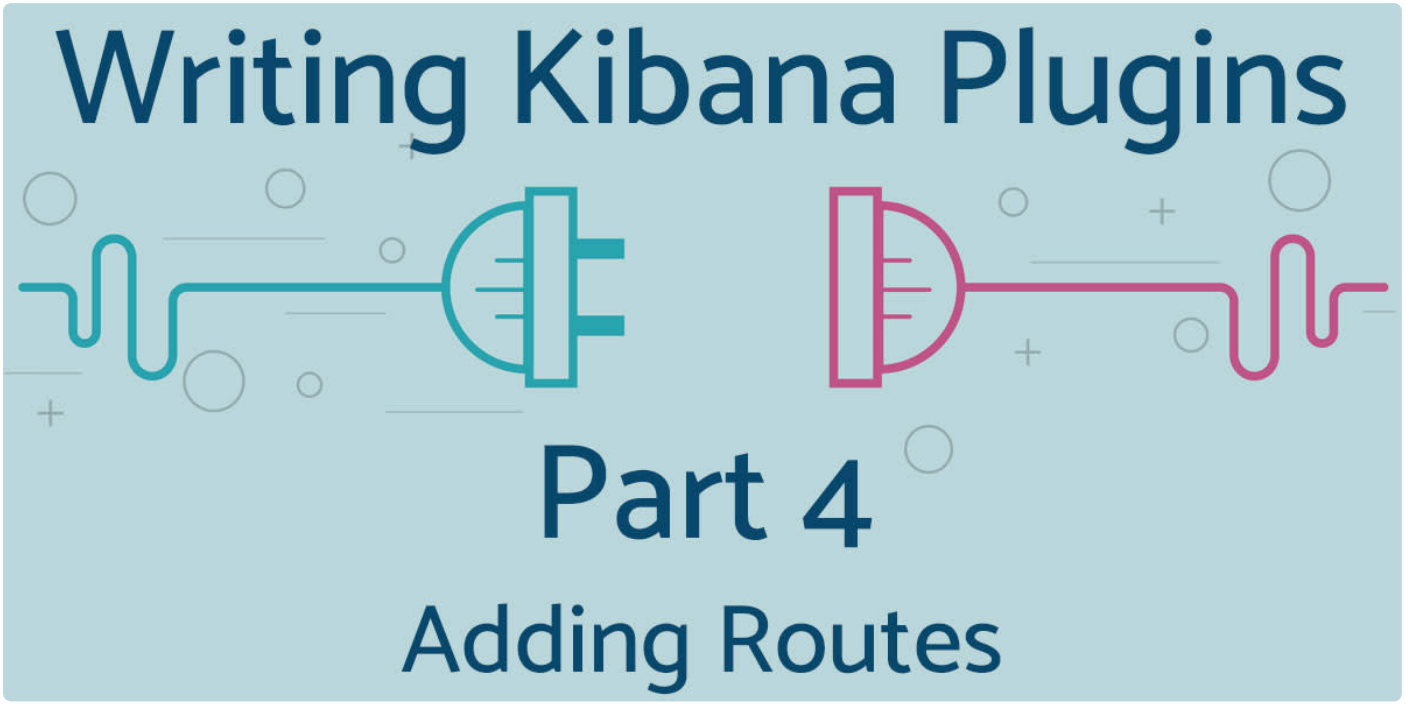
Writing Kibana Plugins, Part 4: Adding Routes

Jochen Kressin · 2023-03-21



In this post, we will add some routes to the server to get, add and delete data to our sample index.

Refresher: Kibana Architecture

As a refresher, we will look briefly at the architecture of Kibana again. It's easy to get confused about what code runs on the server and the client.

Kibana has a client and a server part. The server is based on Node.js and provides the business logic. The client is based on React.js and runs in the browser. The client uses the business logic by calling routes that the server offers.

We want to add new routes to the server for our demo plugin, making it possible for the client to fetch, update, create, and delete data in our sample index. We will implement the client code in this series's next and last article.

Routes in Kibana

Navigation in Kibana is implemented by adding routes. Routes can be added on the server side and on the client side. The main difference is that:

- client-side routes typically communicate with the Kibana server, normally to call endpoints registered by the same plugin
- server-side routes allow plugins to register endpoints that can be called from the client-side

© 2026 floragunn GmbH · search-guard.com

The Router Object

In Kibana, server-side routes are registered using a Router object that can be obtained from the HttpService. This is typically done in the *setup()* lifecycle method of our plugin class.

We will define the routes in a separate file called *server/routes/index.js* for better readability. The *defineRoutes()* method in this file is then called in the *start()* lifecycle method:

server/plugins.ts

```
import {
  PluginInitializerContext,
  CoreSetup,
  CoreStart,
  Plugin,
  Logger,
} from '../../../src/core/server';

...
import { defineRoutes } from './routes';
...

export class SgKibanaDemoPluginPlugin
  implements Plugin<SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart> {

  private readonly logger: Logger;

  constructor(initializerContext: PluginInitializerContext) {
    this.logger = initializerContext.logger.get();
  }

  public setup(core: CoreSetup) {
    this.router = core.http.createRouter();
    return {};
  }

  public start(core: CoreStart) {
    // Register server-side routes
    defineRoutes({
      logger: this.logger,
      router: this.router,
      clusterClient: core.elasticsearch.client,
    });
  }

  ...

}
```

server/routes/index.js

```
import { schema } from '@kbn/config-schema';

export function defineRoutes({ router, logger, clusterClient }) {
}
```

We also import a schema object from *@kbn/config-schema*. This object can later be used to validate the request body of the route calls.

Registering a Route

Routes are registered by calling one of the *get*, *put*, *post* or *delete* methods on the router object. We need to specify the route path, the handler, and, optionally, validation for the request's body, if any. Let's have a look at an example which implements a route that adds a new account document to our index:

```
router.post(
  {
    path: '/api/sg_kibana_demo_plugin/accounts',
    validate: {
      body: schema.object({
        name: schema.string(),
      }),
    }
  },
  (context, req, res) => {
    return res.ok({
      body: {
        message: `ok`
      }
    });
  }
)
```

This route can now be called from the client side. It validates that the request body contains a field called *name* of type *String*. It finally returns a JSON response body.

```
{message: "ok"}
```

Separate Routes and Handlers

To improve readability and to ensure separation of concerns, it is often good practice to separate the routes definition from the actual handlers. In the example above, we do this in a specific function instead of directly implementing the handler logic in the routes definition. This function can also be moved to a separate file.

© 2026 floragunn GmbH · search-guard.com

```
router.post(
  {
    path: '/api/sg_kibana_demo_plugin/accounts',
    validate: {
      body: schema.object({
        name: schema.string(),
      }),
    },
  },
  addAccount({ clusterClient, logger })
);

...

function addAccount({ clusterClient, logger }) {
  return async function (context, request, response) {
     ...
  };
}
```

Calling Elasticsearch in a Route Handler

To complete the example above, we now want to call the Elasticsearch index API endpoint to add the document in the request body to our index. For this, we use the Elasticsearch JavaScript client, which is exposed by the *clusterClient* object.

Reminder: We have passed this object to the *defineRoutes* function from our server plugin class's *start()* method. If you want to learn more about the client object and its usage, head over to the previous article in this series.

The complete code example for the *post* route looks like this:

```
import { schema } from '@kbn/config-schema';

export function defineRoutes({ router, logger, clusterClient }) {

  router.post(
    {
    path: '/api/sg_kibana_demo_plugin/accounts',
    validate: {
      body: schema.object({
        name: schema.string(),
      }),
    },
    addAccount({ clusterClient, logger })
  );

}

function addAccount({ clusterClient, logger }) {
  return async function (context, request, response) {
    try {
      const { body } = request;
      const resp = await clusterClient
        .asScoped(request)
        .asInternalUser.index({ body, index: 'searchguard_demo_accounts', refresh: true });
      return response.ok({ body: resp });
    } catch (error) {
      logger.error(`Add account error. ${error.stack}`);
      return response.internalError({ body: error });
    }
  };
}
```

Path Parameters

When implementing REST-like routes, we also need access to path parameters. For example, if we want to delete or get a specific account like:

```
DELETE /api/sg_kibana_demo_plugin/accounts/1234
```

Let's have a look at how we can specify a corresponding route and handler:

```
router.delete(
  {
    path: `/api/sg_kibana_demo_plugin/accounts/{id}`,
    validate: {
      params: schema.object({
        id: schema.string(),
      }),
    },
  },
  deleteAccount({ clusterClient, logger })
);
```

© 2026 floragunn GmbH · search-guard.com

```
function deleteAccount({ clusterClient, logger }) {
  return async function (context, request, response) {
    try {
      const {
        params: { id },
      } = request;

      const resp = await clusterClient
        .asScoped(request)
        .asInternalUser.delete({ id, index: 'searchguard_demo_accounts', refresh: true });

      return response.ok({ body: resp });
    } catch (error) {
      logger.error(`Delete account. ${error.stack}`);
      return response.internalError({ body: error });
    }
  };
}
```

We leave it up to fellow readers to implement the missing routes and methods for getting all accounts and a single account.

Where to go Next

- Writing Kibana Plugins, part 1: Plugin Basics
- Writing Kibana Plugins, part 2: A Skeleton Plugin
- Writing Kibana Plugins, part 3: Calling the Elasticsearch API
- 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-part-4/
Search Guard is Security and Alerting for Elasticsearch and Kibana — search-guard.com · Start a free trial