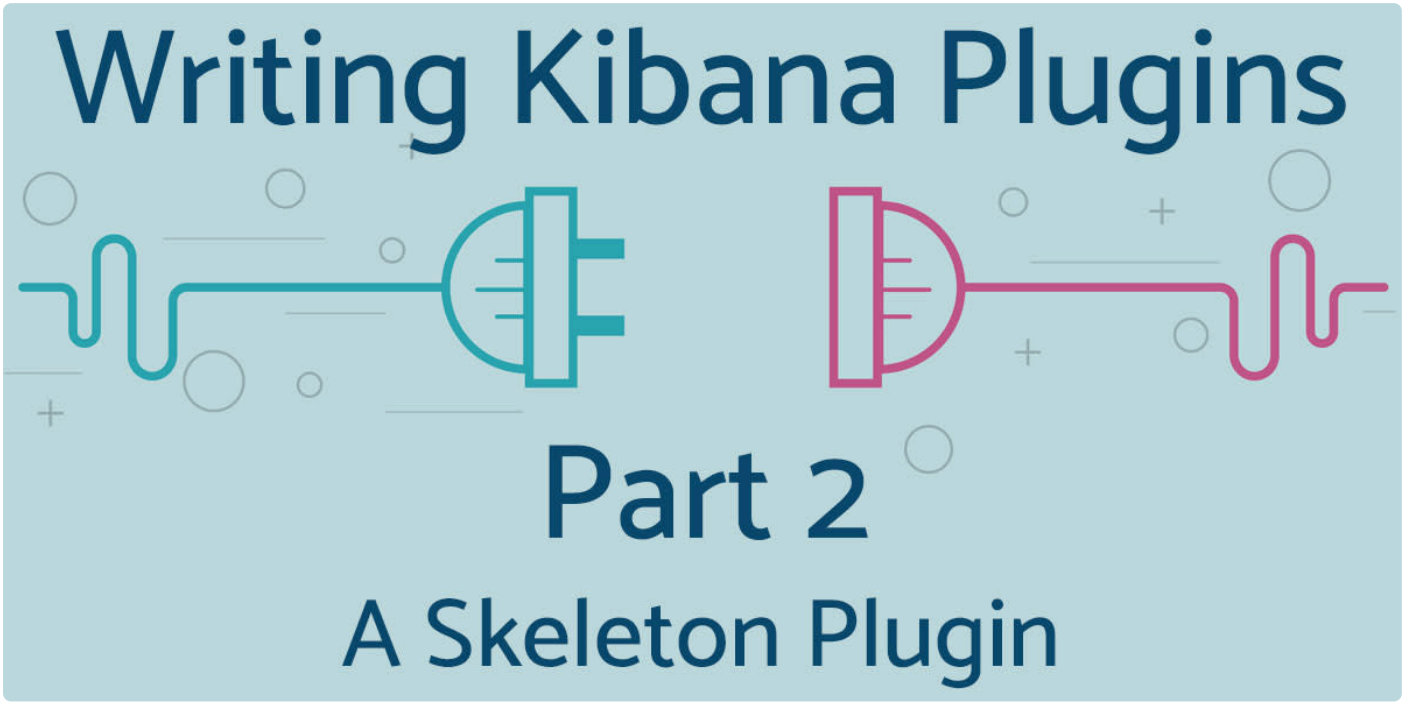
Writing Kibana Plugins - Part 2: A Skeleton Plugin

Jochen Kressin · 2022-12-08



In the last part of this article, we have set up a Kibana plugin skeleton and displayed it in the left navigation bar. This part will look at the various files and artifacts that make up a typical Kibana plugin.

Metadata

The plugin's metadata is located in two files: kibana.json and package.json.

The package.json is a standard file of npm and does not contain many surprises:

```
$ cat package.json
{
  "name": "sgKibanaDemoPlugin",
  "version": "1.0.0",
  "private": true,
  "scripts": {
    "build": "yarn plugin-helpers build",
    "plugin-helpers": "node ../../scripts/plugin_helpers",
    "kbn": "node ../../scripts/kbn"
  }
}
```

It contains the script instructions that you need to build the plugin. If you have any dependencies to other modules, either for runtime or development, you declare them here.

The kibana.json, as the name implies, is specific

```
$ cat kibana.json
{
  "id": "sgKibanaDemoPlugin",
  "version": "1.0.0",
  "kibanaVersion": "7.17.7",
  "server": true,
  "ui": true,
  "requiredPlugins": ["navigation"],
  "optionalPlugins": []
}
```

Note that we declare two versions here: "version" means the version of your plugin, while "kibanaVersion" refers to the Kibana version your plugin is built for.

Kibana (and Elasticsearch) have a strict version check. For example, if kibana.json states that the Kibana version is 7.17.7, you cannot run your plugin with Kibana 7.17.6. In other words, you need to build a separate version of your plugin for each Kibana version, even if it is just a Kibana patch release!

You also need to specify whether your plugin is a server plugin, a UI plugin, or both. A typical Kibana plugin will have both a UI and a server part, so we declare:

```
  "server": true,
  "ui": true,
```

Last, you can declare plugin dependencies in the requiredPlugins and the optionalPlugins. Both take an array of plugin ids. You can then refer to these plugins later in your code.

```
  "requiredPlugins": ["navigation"],
  "optionalPlugins": []
```

UI and Server Plugins

The distinction between UI and server plugins is a core architectural element of Kibana. In short:

A UI plugin only adds display logic to Kibana and uses existing backend logic and routes. For example, If you want to write a plugin that fetches data from Elasticsearch and displays it in a table, you only need a UI plugin.

UI plugins are React.js applications written in either JavaScript or TypeScript. They typically use the Elastic UI Framework (EUI) design library for widgets and components.

A server plugin adds backend logic and routes to the Kibana server that UI plugins can then call. For example, if you run an Elasticsearch plugin like SearchGuard that adds new API endpoints, you need to write a Kibana server plugin to access these endpoints. This is done by declaring routes and handlers. We will cover that later.

Initialization

Both UI and server plugins have a well defined entry point for initialization purposes:

server/index.ts

```
import { PluginInitializerContext } from '../../../src/core/server';
import { SgKibanaDemoPluginPlugin } from './plugin';

//  This exports static code and TypeScript types,
//  as well as the Kibana Platform `plugin()` initializer.
export function plugin(initializerContext: PluginInitializerContext) {
  return new SgKibanaDemoPluginPlugin(initializerContext);
}

export { SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart } from './types';
```

The *initializerContext* object holds the Kibana core context, a set of shared services available for all plugins.

public/index.ts

```
import './index.scss';

import { SgKibanaDemoPluginPlugin } from './plugin';

// This exports static code and TypeScript types,
// as well as, Kibana Platform `plugin()` initializer.
export function plugin() {
  return new SgKibanaDemoPluginPlugin();
}
export { SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart } from './types';
```

Lifecycle Methods

UI and server plugins should implement the three lifecycle methods: setup, start, and stop.

```
   public setup(core: CoreSetup) {
       // register stuff
       return {};
   }

   public start(core: CoreStart) {
       // execute stuff
       return {};
   }

   public stop() {
     // stop stuff, e.g. subscriptions
   }
```

Code that must be executed while the plugin is running must be inside the start method. If you need to stop some processes explicitly on the plugin stop, put it in the stop method.

It's important to keep the lifecycle in mind while developing because Kibana passes a different set of services as the core method argument. For example, Kibana passes a service to register a UI application in the setup method but doesn’t pass it to other methods.

Each method can return an object. Use this to expose the plugin functionality to other plugins.

Lifycycle Methods in Server Plugins

This is a typical example of how to use the lifecycle methods for a server plugin. The constructor is called with an initializerContext object when Kibana instantiates the plugin. It consists of environment data, logger, config, and config helpers.

The setup method has a CoreSetup object as argument. It consists of the Kibana settings service, HTTP server service, Elasticsearch client, status observables, capabilities service, and other services. This is where you typically register new routes by using the HTTP router service.

The CoreStart object, which is passed to the start method, consists of a limited subset of the setup set of services that can be used during the plugin runtime

server/plugin.ts

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

import { SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart } from './types';
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.logger.debug('sgKibanaDemoPlugin: Setup');
    const router = core.http.createRouter();

    // Register server side APIs
    defineRoutes(router);

    return {};
  }

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

  public stop() {}
}
```

Lifycycle Methods in UI Plugins

You can find the same lifecycle methods in UI plugins.

public/plugin.ts

```
import { i18n } from '@kbn/i18n';
import { AppMountParameters, CoreSetup, CoreStart, Plugin } from '../../../src/core/public';
import {
  SgKibanaDemoPluginPluginSetup,
  SgKibanaDemoPluginPluginStart,
  AppPluginStartDependencies,
} from './types';
import { PLUGIN_NAME } from '../common';

export class SgKibanaDemoPluginPlugin
  implements Plugin<SgKibanaDemoPluginPluginSetup, SgKibanaDemoPluginPluginStart> {

  public setup(core: CoreSetup): SgKibanaDemoPluginPluginSetup {
    // Register an application into the side navigation menu
    core.application.register({
      id: 'sgKibanaDemoPlugin',
      title: PLUGIN_NAME,
      async mount(params: AppMountParameters) {
        // Load application bundle
        const { renderApp } = await import('./application');
        // Get start services as specified in kibana.json
        const [coreStart, depsStart] = await core.getStartServices();
        // Render the application
        return renderApp(coreStart, depsStart as AppPluginStartDependencies, params);
      },
    });

    // Return methods that should be available to other plugins
    return {
      getGreeting() {
        return i18n.translate('sgKibanaDemoPlugin.greetingText', {
          defaultMessage: 'Hello from {name}!',
          values: {
            name: PLUGIN_NAME,
          },
        });
      },
    };
  }

  public start(core: CoreStart): SgKibanaDemoPluginPluginStart {
    return {};
  }

  public stop() {}
}
```

Don't worry about the details here, we will dig into it when we stitch everything together i the next article.

Ok, that was a lot of theory. What now?

Yes, it was, but don't worry. In the following article, we will cover a full round-trip where we index some random data, create new routes to fetch and massage the data and then display it. See you then!

Where to go next

- Writing Kibana Plugins, part 1: Plugin Basics
- Writing Kibana Plugins, part 3: Calling the Elasticsearch API
- Writing Kibana Plugins, part 4: Adding Routes
- 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-2/