Getting Started with the OpenFeature Java SDK and Spring Boot

Introduction

This walk-through teaches you the basics of using OpenFeature in Java in the context of a Spring Boot Web application.

You'll learn how to:

- Integrate the OpenFeature Java SDK
- Install and configure the OpenFeature provider
- Perform basic feature flagging

Requirements

This walk-through assumes that:

- You have a basic knowledge of Java and Spring Boot
- You have Java 8 or later
- You have Docker installed and running on the host system

Walk-through

Step 1: Create a Spring Boot Web application

To get started, visit Spring Initializer. This website allows you to download a pre-configured Spring boot application.

For our use case, add **Spring Web** from the dependencies section and set the following **project metadata** values,

- Group: com
- Artifact: demo

Select a Java version matching your installation and download the application using the **GENERATE** button. Extract the archive and open it in your favorite editor.

Step 2: Add dependencies

Based on the selected build system for Spring Boot application, add OpenFeature Java SDK and flagd provider dependencies to existing dependency management configuration.

Given below are dependencies for Maven and Gradle with the latest dependency versions at the time of writing this walk-through.

Maven Gradle

```
<dependency>
    <groupId>dev.openfeature</groupId>
    <artifactId>sdk</artifactId>
    <version>1.8.0</version>
</dependency>
<dependency>
    <groupId>dev.openfeature.contrib.providers</groupId>
    <artifactId>flagd</artifactId>
    <version>0.8.0</version>
</dependency>
```

Step 3: Add code

The `OpenFeatureAPI` class is the main access point to OpenFeature SDK. This class is designed to act as a singleton. Let's define a simple bean configuration class called `OpenFeatureBeans`, that allows us to inject the `OpenFeatureAPI` singleton into desired Spring components.

Create the Java class `OpenFeatureBeans.java` inside the package `com.demo` and add the following code,

```java
package com.demo;

import dev.openfeature.sdk.OpenFeatureAPI;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OpenFeatureBeans {

    @Bean
    public OpenFeatureAPI OpenFeatureAPI() {
        final OpenFeatureAPI openFeatureAPI = OpenFeatureAPI.getInstance();

        return openFeatureAPI;
    }
}
```

Then we can add the REST endpoint definition of `/hello` endpoint, along with dependency injection for `OpenFeatureAPI` instance.

Create the Java class `RestHello.java` inside the package `com.demo` and add the following code.

```java
package com.demo;

import dev.openfeature.sdk.Client;
import dev.openfeature.sdk.OpenFeatureAPI;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class RestHello {
    private final OpenFeatureAPI openFeatureAPI;

    @Autowired
    public RestHello(OpenFeatureAPI OFApi) {
        this.openFeatureAPI = OFApi;
    }

    @GetMapping("/hello")
    public String getHello() {
        final Client client = openFeatureAPI.getClient();

        // Evaluate welcome-message feature flag
        if (client.getBooleanValue("welcome-message", false)) {
            return "Hello, welcome to this OpenFeature-enabled website!";
        }

        return "Hello!";
    }
}
```

At this point, we are ready to run the initial version of our application.

Step 4: Run the initial application

Let's compile and run the application.

Maven Gradle

```
mvn clean install
java -jar target/demo-0.0.1-SNAPSHOT.jar
```

Now you can visit the url http://localhost:8080/hello and observe the message **Hello!**.

"Why I'm I seeing that value?", you may ask. Well, it's because a provider hasn't been configured yet. Without a provider to actually evaluate flags, OpenFeature will return the default value. In the next step, you'll learn how to add a provider.

Step 5: Configure a provider (flagd)

Providers are an important concept in OpenFeature because they are responsible for the flag evaluation itself. As we saw in the previous step, OpenFeature without a provider always returns the default value. If we want to actually perform feature flagging, we'll need to register a provider.

Create a new file named `flags.flagd.json` and add the following JSON. Notice that there's a flag called `welcome-message` which matches the flag key referenced earlier. The `welcome-message` flag has `on` and `off` variants that return `true` and `false` respectively. The state property controls whether the feature flag is active or not. Finally, the defaultVariant property controls the variant that should be returned. In this case, the defaultVariant is `off`, therefore the value `false` would be returned.

```
{
  "flags": {
    "welcome-message": {
      "variants": {
        "on": true,
        "off": false
      },
      "state": "ENABLED",
      "defaultVariant": "off"
    }
  }
}
```

> NOTE: This configuration is specific for flagd and varies across providers.

With the flagd configuration in place, start flagd service with the following docker command.

> NOTE: On Windows WSL is required both for running docker and to store the file. This is a limitation of Docker (https://github.com/docker/for-win/issues/8479)

```
docker run -p 8013:8013 -v $(pwd)/:/etc/flagd/ -it ghcr.io/open-feature/flagd:latest start --
uri file:/etc/flagd/flags.flagd.json
```

Finally, let's add the required code change to `OpenFeatureBeans.java` class.

```
package com.demo;

 import dev.openfeature.contrib.providers.flagd.FlagdProvider;
 import dev.openfeature.sdk.exceptions.OpenFeatureError;
import dev.openfeature.sdk.OpenFeatureAPI;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class OpenFeatureBeans {

    @Bean
    public OpenFeatureAPI OpenFeatureAPI() {
        final OpenFeatureAPI openFeatureAPI = OpenFeatureAPI.getInstance();

         // Use flagd as the OpenFeature provider and use default configurations
         try {
             openFeatureAPI.setProviderAndWait(new FlagdProvider());
         } catch (OpenFeatureError e) {
             throw new RuntimeException("Failed to set OpenFeature provider", e);
         }

        return openFeatureAPI;
    }
}
```

Step 6: Rerun the application

Let's rerun our Java Spring Boot application.

Maven Gradle

```
mvn clean install
java -jar target/demo-0.0.1-SNAPSHOT.jar
```

Revisit the endpoint http://localhost:8080/hello and you will see the same value.

Let's change the feature flag in our `flags.flagd.json`, making `defaultVariant` to `on`

```
{
  "flags": {
    "welcome-message": {
      "variants": {
        "on": true,
        "off": false
      },
      "state": "ENABLED",
       "defaultVariant": "off"
       "defaultVariant": "on"
    }
  }
}
```

Revisit the endpoint http://localhost:8080/hello and you will be greeted with `Hello, welcome to this OpenFeature-enabled website!`

Conclusion

This walk-through introduced you to the OpenFeature Java SDK and how it can be easily integrated into well-known frameworks such as Spring Boot. It covered how a provider can be configured to perform the flag evaluation and introduced basic feature flagging concepts. It also showcased how feature flags can be updated at runtime, without requiring a code change and a redeployment.

Edit this page