1. Overview

Spring Boot Admin is a web application, used for managing and monitoring Spring Boot applications. Each application is considered as a client and registers to the admin server. Behind the scenes, the magic is given by the Spring Boot Actuator endpoints.

In this article, we're going to describe steps for configuring a Spring Boot Admin server and how an application becomes a client.

2. Admin Server Setup

First of all, we need to create a simple Spring Boot web application and also add the following Maven dependency:

```
<dependency>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-starter-server</artifactId>
<version>2.2.2</version>
</dependency>
```

After this, the *@EnableAdminServer* will be available, so we'll be adding it to the main class, as shown in the example below:

```
@EnableAdminServer
@SpringBootApplication
public class SpringBootAdminServerApplication {
public static void main(String[] args) {
SpringApplication.run(SpringBootAdminServerApplication.class, args);
}
}
```

At this point, we're ready to start the server and register client applications.

3. Setting Up a Client

Now, after we've set up our admin server, we can register our first Spring Boot application as a client. We must add the following Maven dependency:

```
<dependency>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-starter-client</artifactId>
```

```
<version>2.2.2</version>
</dependency>
```

Next, we need to configure the client to know about the admin server's base URL. For this to happen, we just add the following property:

```
spring.boot.admin.client.url=http://localhost:8080
```

Starting with Spring Boot 2, endpoints other than *health* and *info* are not exposed by default.

Let's expose all the endpoints:

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

4. Security Configuration

The Spring Boot Admin server has access to the application's sensitive endpoints, so **it's advised that we add some security configuration to both admin and client application.**

At first, we'll focus on configuring the admin server's security. We must add the following Maven dependencies:

```
<dependency>
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-server-ui-login</artifactId>
<version>1.5.7</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-security</artifactId>
<version>2.1.8.RELEASE</version>
</dependency>
```

This will enable security and add a login interface to the admin application.

Next, we'll add a security configuration class as you can see below:

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
private final AdminServerProperties adminServer;
public WebSecurityConfig(AdminServerProperties adminServer) {
this.adminServer = adminServer;
}
@Override
protected void configure(HttpSecurity http) throws Exception {
SavedRequestAwareAuthenticationSuccessHandler successHandler =
```

```
new SavedRequestAwareAuthenticationSuccessHandler();
successHandler.setTargetUrlParameter("redirectTo");
successHandler.setDefaultTargetUrl(this.adminServer.getContextPath() + "/");
http
.authorizeRequests()
.antMatchers(this.adminServer.getContextPath() + "/assets/**").permitAll()
.antMatchers(this.adminServer.getContextPath() + "/login").permitAll()
.anyRequest().authenticated()
.and()
.formLogin()
.loginPage(this.adminServer.getContextPath() + "/login")
.successHandler(successHandler)
.and()
.logout()
.logoutUrl(this.adminServer.getContextPath() + "/logout")
.and()
.httpBasic()
.and()
.csrf()
.csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
.ignoringRequestMatchers(
new AntPathRequestMatcher(this.adminServer.getContextPath() +
"/instances", HttpMethod.POST.toString()),
new AntPathRequestMatcher(this.adminServer.getContextPath() +
"/instances/*", HttpMethod.DELETE.toString()),
new AntPathRequestMatcher(this.adminServer.getContextPath() + "/actuator/**"))
.and()
.rememberMe()
.key(UUID.randomUUID().toString())
.tokenValiditySeconds(1209600);
}
}
```

There's a simple security configuration, but after adding it, we'll notice that the client cannot register to the server anymore.

In order to register the client to the newly secured server, we must add some more configuration into the property file of the client:

```
spring.boot.admin.client.username=admin
spring.boot.admin.client.password=admin
```

We're at the point, where we secured our admin server. In a production system, naturally, the applications we're trying to monitor will be secured. So, we'll add security to the client as well – and we'll notice in the UI interface of the admin server that the client information is not available anymore.

We have to add some metadata that we'll send to the admin server. This information is used by the server to connect to client's endpoints:

```
spring.security.user.name=client
spring.security.user.password=client
spring.boot.admin.client.instance.metadata.user.name=${spring.security.user.name}
spring.boot.admin.client.instance.metadata.user.password=${spring.security.user.password}
```

Sending credentials via HTTP is, of course, not safe – so the communication needs to go over HTTPS.

5. Monitoring and Management Features

Spring Boot Admin can be configured to display only the information that we consider useful. We just have to alter the default configuration and add our own needed metrics:

```
spring.boot.admin.routes.endpoints=env, metrics, trace, jolokia, info, configprops
```

As we go further, we'll see that there are some other features that can be explored. We're talking about *JMX bean management* using *Jolokia* and also *Loglevel* management.

Spring Boot Admin also supports cluster replication using Hazelcast. We just have to add the following Maven dependency and let the autoconfiguration do the rest:

```
<dependency>
<groupId>com.hazelcast</groupId>
<artifactId>hazelcast</artifactId>
<version>3.12.2</version>
</dependency>
```

If we want a persistent instance of Hazelcast, we're going to use a custom configuration:

```
@Configuration
public class HazelcastConfig {
@Bean
public Config hazelcast() {
MapConfig eventStoreMap = new MapConfig("spring-boot-admin-event-store")
.setInMemoryFormat(InMemoryFormat.OBJECT)
.setBackupCount(1)
.setEvictionPolicy(EvictionPolicy.NONE)
.setMergePolicyConfig(new
MergePolicyConfig(PutIfAbsentMapMergePolicy.class.getName(), 100));
MapConfig sentNotificationsMap = new MapConfig("spring-boot-admin-application-
store")
```

```
.setInMemoryFormat(InMemoryFormat.OBJECT)
.setBackupCount(1)
.setEvictionPolicy(EvictionPolicy.LRU)
.setMergePolicyConfig(new
MergePolicyConfig(PutIfAbsentMapMergePolicy.class.getName(), 100));
Config config = new Config();
config.addMapConfig(eventStoreMap);
config.addMapConfig(sentNotificationsMap);
config.setProperty("hazelcast.jmx", "true");
config.getNetworkConfig()
.getJoin()
.getMulticastConfig()
.setEnabled(false);
TcpIpConfig tcpIpConfig = config.getNetworkConfig()
.getJoin()
.getTcpIpConfig();
tcpIpConfig.setEnabled(true);
tcpIpConfig.setMembers(Collections.singletonList("127.0.0.1"));
return config;
}
}
```

6. Notifications

Next, let's discuss the possibility to receive notifications from the admin server if something happens with our registered client. The following notifiers are available for configuration:

- Email
- PagerDuty
- OpsGenie
- Hipchat
- Slack
- Let's Chat

6.1. Email Notifications

We'll first focus on configuring mail notifications for our admin server. For this to happen, we have to add the mail starter dependency as shown below:

```
<dependency>
```

```
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-mail</artifactId>
<version>2.1.7.RELEASE</version>
</dependency>
```

After this, we must add some mail configuration:

```
spring.mail.host=smtp.example.com
spring.mail.username=smtp_user
spring.mail.password=smtp_password
spring.boot.admin.notify.mail.to=admin@example.com
```

Now, whenever our registered client changes his status from UP to OFFLINE or otherwise, an email is sent to the address configured above. For the other notifiers, the configuration is similar.

6.2. Hipchat Notifications

As we'll see, the integration with Hipchat is quite straightforward; there are only a few mandatory properties to set:

```
spring.boot.admin.notify.hipchat.auth-token=<generated_token>
spring.boot.admin.notify.hipchat.room-id=<room-id>
spring.boot.admin.notify.hipchat.url=https://yourcompany.hipchat.com/v2/
```

Having these defined, we'll notice in the Hipchat room that we receive notifications whenever the status of the client changes.

6.3. Customized Notifications Configuration

We can configure a custom notification system having at our disposal some powerful tools for this. We can use a *reminding notifier* to send a scheduled notification until the status of client changes.

Or maybe we want to send notifications to a filtered set of clients. For this, we can use a *filtering notifier:*

```
@Configuration
public class NotifierConfiguration {
private final InstanceRepository repository;
private final ObjectProvider<List<Notifier>> otherNotifiers;
public NotifierConfiguration(InstanceRepository repository,
ObjectProvider<List<Notifier>> otherNotifiers) {
this.repository = repository;
this.otherNotifiers = otherNotifiers;
}
@Bean
```

```
public FilteringNotifier filteringNotifier() {
CompositeNotifier delegate =
new CompositeNotifier(this.otherNotifiers.getIfAvailable(Collections::emptyList));
return new FilteringNotifier(delegate, this.repository);
}
@Bean
public LoggingNotifier notifier() {
return new LoggingNotifier(repository);
}
@Primary
@Bean(initMethod = "start", destroyMethod = "stop")
public RemindingNotifier remindingNotifier() {
RemindingNotifier remindingNotifier = new RemindingNotifier(filteringNotifier(),
repository);
remindingNotifier.setReminderPeriod(Duration.ofMinutes(5));
remindingNotifier.setCheckReminderInverval(Duration.ofSeconds(60));
return remindingNotifier;
}
}
```

7. Conclusion

This intro tutorial covers the simple steps that one has to do, in order to monitor and manage his Spring Boot applications using Spring Boot Admin.

The autoconfiguration permits us to add only some minor configurations and at the end, to have a fully working admin server.

And, as always, the sample code of this guide can be found over on Github.