**Monitoramento e documentação**

**Monitoramento com Spring Boot Actuator**

Na última aula nós terminamos a parte de segurança utilizando token. Na aula de hoje vamos aprender um pouco sobre monitoramento. Se você tem uma API REST que tem dezenas, centenas de clientes é importante ter um monitoramento, ainda mais se você estiver trabalhando com arquitetura de micro serviços, em que você tem vários serviços que você tem que gerenciar.

[00:40] Pensando nisso, o pessoal do Spring Boot criou um projeto chamado actuator. É uma API que tem endpoints que devolvem um JSON com informações sobre a API. É uma facilidade para conseguir monitorar e acompanhar o andamento da nossa API.

[01:12] Como ele é um módulo, vamos precisar adicioná-lo ao nosso projeto. Precisamos adicionar a dependência do Spring Boot actuator. Vou salvar. O Maven vai baixar as dependências. Só de ter o actuator como dependência do projeto, ele já vem habilitado por padrão.

[01:45] Vou rodar nossa aplicação. Por padrão, ele vai criar um novo endpoint na nossa aplicação, que é o /actuator. Se dispararmos uma requisição GET para /actuator, ele vai devolver um JSON com algumas informações sobre a aplicação. Vou testar pelo Postman mesmo. Ele me deu um 403, porque como está com Spring security, é um novo endereço, precisamos liberar o acesso naquele método onde configuramos as URLs.

[02:54] Para testes, vou colocar um permitAll, mas quando você for colocar sua API em produção, a ideia é que você remova isso, porque esse endpoint devolve informações sensíveis sobre a aplicação, você não quer deixar isso aberto para qualquer pessoa. Vai ser só para sua equipe, a equipe de infraestrutura da sua empresa.

[03:25] Vou disparar de novo a requisição. Ele me dá um 200 e devolve um JSON com alguns links, que são sub-recursos de coisas que você pode acessar do actuator. Por exemplo, o self, o health, para saber informações sobre a saúde da sua API, o info, que devolve informações sobre a API.

[04:07] Vamos disparar uma requisição para o /health. Deu proibido, porque esse é um sub-endereço. Vou voltar no configuration e colocar /actuator/\*\*. Salvo e testo de novo. A única coisa que o health me devolve é um “up”, indicando que a aplicação está de pé.

[04:47] Vamos testar o /info. Vem vazio, porque por padrão ele não devolve nada. Conseguimos configurar no Spring para ele devolver informações da API e para expor outros endpoints. Vamos fazer isso. Como é uma configuração, fica no application.properties. Eu vou colar algumas propriedades. Você não precisa decorar isso. No exercício você vai pegar esse arquivo pronto. Mas só para entender, na primeira linha estou dizendo para o health sempre mostrar todos os detalhes, não só o up. Na segunda, quero que ele inclua outros endpoints com outras informações sobre a nossa API. E as duas últimas servem para exibir informações sobre nossa aplicação.

[06:15] No valor, passamos a sintaxe @project.name@. Isso é para falar para o Spring Boot que é para puxar as informações do arquivo .xml. No .xml, no topo do arquivo, tem as informações de groupId, artifactId, etc. Conseguimos falar para ele puxar todas essas informações e jogar no actuator.

[06:52] Vou salvar, ele vai reiniciar. Vou disparar de novo a requisição para o /info. Agora veio com as informações que coloquei para ele expor no application properties. Vamos disparar a requisição para o health. Ao invés de devolver só o status up ou down, ele devolve mais alguns detalhes. São informações que conseguimos descobrir, habilitar, para expor no endpoint do actuator.

[08:04] Tudo que ele devolve é no formato JSON. O actuator não tem interface gráfica, não mostra uma tela, um dashboard com gráficos e coisas do gênero. Ele só vai devolver o JSON com essas informações. Então, na verdade, a equipe de desenvolvimento da API só vai ativar o actuator, configurar o que queremos e não queremos expor, e passar o endereço para o pessoal que vai cuidar da parte de infraestrutura. Esse pessoal já tem ferramentas que vão disparar requisições para esse endereço, vão puxar o JSON, montar um gráfico, algo do gênero.

[08:52] No geral, na área de TI você tem o pessoal responsável pela parte de operações. Eles é que vão consumir o endpoint para fazer o monitoramento, configurar alertas, e-mails e coisas do gênero. A única coisa que nós fazemos é liberar o actuator para expor as informações da nossa API. Nós não precisamos implementar essas informações na mão. O próprio actuator já tem a lógica que expõe as informações automaticamente.

[09:20] Esse é o actuator. É uma maneira de você expor informações da sua API para que alguém consiga puxar esses dados e fazer o monitoramento. No próximo vídeo vou mostrar uma ferramenta que podemos utilizar para consumir essas informações e exibir uma tela com gráficos, de maneira visual, caso sua equipe de operações não saiba como consumir. Ou você mesmo pode usar para fazer o monitoramento, ter uma interface gráfica de administração dessas informações.

# actuator

management.endpoint.health.show-details=always

management.endpoints.web.exposure.include=\*

info.app.name=@project.name@

info.app.description=@project.description@

info.app.version=@project.version@

info.app.encoding=@project.build.sourceEncoding@

info.app.java.version=@java.version@

**Conhecendo Spring Boot Admin**

Na última aula nós adicionamos o Spring Boot actuator ao projeto, habilitamos e fizemos algumas configurações. Conseguimos testar pelo Postman disparando requisições para o endereço do actuator e dando uma olhada no JSON.

[00:20] Como eu tinha dito, a ideia era disponibilizar isso para o time de operações, infraestrutura criar, pegar algum projeto para consumir esse JSON e exibir numa ferramenta. Mas, caso eles não usem, não tenham uma ferramenta específica para isso, existe uma ferramenta que se integra com o Spring Boot, bem bacana, que tem uma interface web para puxar essas informações do actuator e exibir de maneira visual, com gráficos, alertas e outras coisas.

[00:55] Essa ferramenta se chama Spring Boot Admin. Essa ferramenta não é do Spring. Quem desenvolveu foi uma empresa alemã chamada codecentric’s. É uma aplicação web, usando Spring Boot, que consegue consumir e monitorar outras aplicações que estejam usando o Spring Boot, com o Spring Boot actuator ativado. Ele te fornece informações sobre jvm, data source, acesso ao banco de dados, hd, informações sobre as requisições web, consegue até te enviar alertas por e-mail, integrando com outras ferramentas.

[01:59] Se você entrar na página dela, você vai achar o guia de referência. Vamos utilizar a versão 2.1.4, que é compatível com o Spring Boot 2.1.4, que é o que estamos utilizando. Se eu clicar no link tem uma documentação explicando como você faz para utilizar. Basicamente, ele fala para você criar uma nova aplicação com Spring Boot no site do Spring initializr, e o único módulo que você precisa ter do Spring Boot é o módulo web. Além disso, você precisa adicionar mais uma dependência, que é o servidor do Spring Boot Admin. Vamos fazer esse primeiro passo. Vamos criar um projeto de monitoramento.

[02:46] Vou entrar naquele site start.spring.io, vou deixar todas as opções marcadas. O group vai ser br.com.alura, e o artifact vai ser Spring-Boot-admin. O único módulo do qual precisamos é o web. Vou gerar o projeto, ele vai baixar o zip, vou descompactar, extrair a pasta do projeto. Antes de importar para o Eclipse, vou editar na mão o arquivo .xml para trocar a versão do Spring Boot, porque ela já atualizou enquanto as gravações do curso ocorriam. Vou colocar 2.1.4, porque senão ele vai baixar todas as dependências do Spring Boot na versão mais nova.

[04:23] Vou até o Eclipse, e preciso importar esse novo projeto. É uma aplicação com Spring Boot. Vou abrir o .xml. A única coisa que tem aqui é o módulo web e o módulo de testes. Se voltarmos para a documentação, ele fala que nesse projeto precisa ter mais uma dependência, que é a dependência do Spring Boot Admin. Vou adicioná-la. Cuidado com as versões.

[05:20] Vou salvar, ele vai baixar. Temos que ir até a classe main e trocar as anotações. Vou copiar as três anotações que ele dá, vou tirar o @SpringBootApplication e substituir.

[06:10] Vou rodar o projeto. É um projeto com Spring Boot, então ele vai rodar o Spring Boot e vai rodar o módulo do Spring Boot Admin server, porque é um projeto do servidor. Vamos fazer um teste entrando no navegador. Ele entra na interface gráfica do Spring Boot Admin.

[06:38] Por enquanto, eu não configurei nenhuma aplicação para ser monitorada por essa, então está tudo vazio. Só tem um problema. Ele está rodando por padrão no 8080, mas o nosso projeto também está no 8080. Isso vai dar conflito. Eu vou parar o projeto, abrir o application.properties, e vou colocar a propriedade server.port=8081. Quero que o projeto do admin rode na 8081.

[07:22] Agora o endereço do servidor do Spring Boot Admin é 8081. Já estou com o projeto do Spring Boot Admin rodando, mas ele não está monitorando nossa aplicação do fórum. No próximo vídeo nós vamos configurar nosso projeto do fórum para que ele seja administrado pela interface do Spring Boot Admin.

**Monitorando a nossa API com Spring Boot Admin**

Já estamos com o projeto do servidor configurado, só que ele ainda não está monitorando nossa API. Na aula de hoje vamos configurar para que o Spring Boot Admin consiga monitorar nossa API usando aquelas informações disponibilizadas pelo actuator.

[00:28] Se voltarmos na documentação do Spring Boot Admin, logo depois dele falar como criar o projeto do servidor, tem uma sessão onde ele fala como registro clientes. Ele basicamente fala para ir ao projeto, na sua API que você quer monitorar, e você tem que adicionar a dependência do Spring Boot Admin starter-client.

[01:01] Eu vou copiar essa dependência e colar no nosso projeto fórum. Embaixo ele até coloca o Spring security, mas é opcional, e no nosso caso já temos. Ele mostra só para reforçar a ideia de que você vai querer configurar a segurança do projeto, porque o Spring Boot Admin exibe informações sensíveis da aplicação.

[01:48] Coloquei então no projeto fórum a dependência do Spring Boot Admin starter client. Vou salvar, ele vai baixar as dependências. Só que não basta só colocar a dependência. Preciso indicar para o projeto fórum onde está o endereço do projeto server.

[02:10] Voltando para a documentação, na sequência ele fala que precisamos habilitar no projeto cliente duas propriedades apontando para o endereço do servidor. A primeira é para passar o endereço do servidor. A segunda é a que já temos no projeto, para o actuator expor todos os endpoints disponíveis. Vou copiar só a de cima, porque a de baixo nós já temos.

[03:00] A propriedade é bem descritiva. Só que no caso não é local host 8080. É 8081. Vou salvar e rodar o projeto fórum application. Vou abrir a interface do admin, e ele até já atualiza sozinho. Ele já detectou que tem uma aplicação sendo monitorada. Você pode monitorar outras aplicações se você quiser. Você vai ter que pegar cada um dos projetos, adicionar um módulo do starter client e adicionar no application properties a propriedade indicando para o endereço do servidor. Vai aparecer todas as aplicações que estão sendo monitoradas.

[04:05] O que posso fazer agora? Posso clicar no Spring-Boot-application, ele vai abrir uma janela. Você pode clicar nessa janela. Mas não clique no link. Ele me joga para outra tela, onde tem todas as informações de monitoramento dessa API. Ele te fala que é uma aplicação com Spring Boot, fala o endereço IP, o endpoint do actuator, tem aquelas informações que tínhamos visto no JSON, fala informações do health, etc. No canto esquerdo tem um menu em que podemos ver algumas métricas, podemos ver coisas do environment, tem alguns logs, informações da jvm, informações de web, até coisas de cache ele registra.

[05:53] Essa é a interface de monitoramento, utilizando o Spring Boot Admin. Se o pessoal da área de operações da sua empresa não tem uma ferramenta ou quer usar outra ferramenta mais interessante para monitorar aplicações Java com Spring Boot, você pode mostrar para eles que tem essa opção.

[06:20] Esse é o Spring Boot Admin. Com isso, configuramos ele para fazer o monitoramento da nossa API. Tem a interface, gráficos, uma sessão onde podemos configurar alertas. Depois que está tudo funcionando, você tem que ir à parte de security configurations e tirar o permitir todos. Você não quer deixar isso público. Você pode configurar a obrigação de estar logado.

[07:17] As configurações do Admin vão ficar no Spring Boot Admin. As configurações do fórum são as que já fizemos. Por isso na documentação ele fala para colocar o Spring security, para proteger essa tela do Spring Boot Admin.

[07:44] Com isso, vimos uma ferramenta bacana, bem interessante, bem popular, para fazer monitoramento das nossas aplicações com Spring Boot. Essa era a nossa aula. No próximo vídeo vamos começar a falar sobre documentação, que é outra coisa importante. Se você tem uma API REST que vai ser consumida por vários clientes é importante ter uma documentação para que esses clientes consigam saber os endereços, os endpoints, os parâmetros.

Como fazer para que o **Spring Boot Admin Server** monitore uma API?

Parte superior do formulário

É necessário adicionar o endereço do projeto **Spring Boot Admin Server** nas configurações da API cliente

O projeto cliente é quem deve conhecer o endereço do projeto **Spring Boot Admin Server**.

Nesta aula, aprendemos que:

* Para adicionar o **Spring Boot Actuator** no projeto, devemos adicioná-lo como uma dependência no arquivo **pom.xml**;
* Para acessar as informações disponibilizadas pelo **Actuator**, devemos entrar no endereço <http://localhost:8080/actuator>;
* Para liberar acesso ao **Actuator** no **Spring Security**, devemos chamar o método .antMatchers(HttpMethod.GET, "/actuator/\*\*");
* Para que o **Actuator** exponha mais informações sobre a API, devemos adicionar as propriedades management.endpoint.health.show-details=always e management.endpoints.web.exposure.include=\* no arquivo **application.properties**;
* Para utilizar o **Spring Boot Admin**, devemos criar um projeto **Spring Boot** e adicionar nele os módulos spring-boot-starter-web e spring-boot-admin-server;
* Para trocar a porta na qual o servidor do **Spring Boot Admin** rodará, devemos adicionar a propriedade server.port=8081 no arquivo **application.properties**;
* Para o **Spring Boot Admin** conseguir monitorar a nossa API, devemos adicionar no projeto da API o módulo spring-boot-admin-client e também adicionar a propriedade spring.boot.admin.client.url=http://localhost:8081 no arquivo **application.properties**;
* Para acessar a interface gráfica do **Spring Boot Admin**, devemos entrar no endereço [http://localhost:8081](http://localhost:8081/).

**Documentação da API com SpringFox Swagger**

 Vimos no último vídeo que é importante documentar nossa API para que os clientes consigam entender, saber como funciona, fazer testes. O Swagger vai ser a ferramenta que vamos utilizar. Mas no caso, como vamos utilizar no nosso projeto? Tem um site chamado SpringFox, que é uma biblioteca que consegue documentar uma API REST utilizando o Swagger, sendo que esse projeto é desenvolvido em Java com Spring. Como nossa API REST foi desenvolvida em Java web com Spring, o SpringFox vai nos ajudar a documentar.

[00:54] Um detalhe importante. O SpringFox tem Spring no nome, mas não é uma biblioteca do Spring. É parecido com aquela outra biblioteca que nós usamos para monitoramento. Foi um pessoal que desenvolveu essa biblioteca baseada no Swagger para documentar aplicações utilizando API REST.

[01:20] Vamos para o Eclipse. O primeiro passo é baixar o SpringFox para o nosso projeto, adicionar como uma dependência no Maven. Temos que trocar o groupId para io.springfox. O artifactId vai ser springfox-swagger2. Como esse não vem do Spring Boot, tenho que dizer qual a versão. A versão que vamos utilizar no curso é a 2.9.2. Além disso, precisamos de outra dependência, porque o Swagger vai gerar a documentação da nossa API, vai ler as classes, os endpoints, e disponibilizar esses endpoints, essa documentação através de um site. Nós vamos ter um endpoint, um endereço onde conseguimos navegar pela documentação. Quem faz essa geração da interface é outra dependência.

[02:40] Essa dependência também é do SpringFox, mas o artifactId é springfox-swagger-ui. Cuidado que esse não tem o 2. A versão é a mesma. Vou salvar, o Maven vai baixar as dependências.

[03:06] Ele não vem habilitado por padrão no projeto. Como de costume, para habilitar coisas no projeto, abrimos nossa classe main e colocamos a anotação @EnableSwagger2.

[03:30] O Swagger precisa que criemos uma classe com algumas configurações, precisamos dizer algumas coisas da nossa API. Por exemplo, qual o pacote raiz onde ele vai começar a ler as classes, porque às vezes não quero que ele leia todas as classes do meu projeto. Qual a url base, porque também posso ter algumas URLs restritas. Dentre outras coisas.

[03:53] Como essas configurações são feitas via código Java, eu vou criar uma nova classe no projeto, e vou jogar essa classe no pacote configure, mas vou criar um subpacote chamado Swagger e uma classe com o nome Swagger configurations.

[04:16] Vou colocar o @Configuration em cima da classe, que é aquela anotação do Spring para carregar essa classe de configuração. Dentro dessa classe, preciso definir um bean e devolver um objeto do tipo docket. Vou chamar o método de forumApi. Esse método tem que estar anotado com o @Bean do Spring, para o Spring saber que estou exportando esse bean, que é o objeto do tipo docket.

[04:51] Dentro do método tenho que instanciar esse objeto docket e setar todas as informações que o SpringFox Swagger precisa para configurar nosso projeto. Vou colar o código, que é um pouco chato, vocês vão pegar ele pronto no exercício.

[05:14] Eu estou retornando um new docket. Na hora de dar new nessa classe você tem que dizer qual o tipo de documentação, que no caso é usando o Swagger2. Na sequência, .select, .apis, e aí tenho que dizer para ele qual o basePackage. A partir de qual pacote ele vai começar a ler as classes do nosso projeto. No nosso caso não tenho restrições, quero que ele leia todas as classes do projeto.

[05:47] Na sequência tenho o .paths, e aí passamos quais endpoints o SpringFox Swagger precisa analisar. No caso, também não tenho nenhuma url restrita, então coloquei /\*\*.

[06:07] Depois temos o .build. No final, só coloquei o detalhe do .ignoredParameterTypes(Usuario.class). Quero que ele ignore todas as URLs que trabalham com a classe usuário, porque senão, na tela do Swagger vai começar a aparecer os dados da senha do usuário. Não quero isso.

[06:42] Já está configurado o Swagger no projeto. Vou rodar nosso fórumApplication, ele vai inicializar. Quando eu rodar o projeto, o Swagger já vai rodar a classe, escanear os pacotes, entrar nos endereços da nossa aplicação, e ele gera essa documentação em um site. Para acessar esse site, entramos no localhost:8080/Swagger-ui.html. Esse é o endereço para abrir a documentação do Swagger.

[07:20] Deu o erro 403, porque como estamos com o Spring security configurado, preciso liberar o endereço do Swagger. Vou abrir a classe do securityConfigurations. Dentro dessa classe temos que liberar quais URLs queremos liberar e quais bloquear. Eu não liberei, ele caiu automaticamente em bloquear. Basta liberar.

[07:49] Tem um problema. Não é só esse endereço que precisamos liberar. Esse é o endereço raiz, mas quando entramos nele, o Swagger dispara algumas requisições para carregar alguns arquivos estáticos, como CSS, Java script e imagens. Vamos precisar liberar vários endereços. Mas não vamos fazer isso aqui.

[08:12] Lembra que tinha um terceiro método configure na classe SecurityConfiguration? Era o que recebe o webSecurity como parâmetro. Nesse método configuro as coisas que não quero que o Spring security intercepte.

[08:28] Quero que o Spring security ignore as requisições para o Swagger. Não é para ele interceptar, ter login, autenticação para acessar a documentação da API. Tenho essa configuração copiada, só vou colar. A ideia é que dentro desse método coloco ignoring().antMatchers, e passo uma lista de string com todas as URLs que queremos ignorar. Precisamos liberar algumas URLs do Swagger. Por exemplo, /\*\*html. São endereços que o Swagger chama quando entramos no navegador, naquela url para abrir a documentação. Depois no exercício vocês vão pegar isso certinho.

[09:28] Feito isso, salvei. Ele vai reiniciar. Acho que já consigo entrar no Swagger.ui.html. Aqui tem a documentação da nossa API. Ele encontrou três controllers. O AutenticacaoController, o HelloController e o TopicosController. Você pode navegar por essa documentação. Posso clicar em um controller e ele vai mostrar todos os métodos que têm mapeamento. Clicando no método, ele mostra todos os parâmetros, quais as respostas possíveis.

[10:50] Em cima, perto da url, tem o botão try it out, posso clicar nele e fazer o teste diretamente do Swagger. Além de ter essa documentação, esse site com um visual bacana, também consigo testar a API. Até se eu tiver uma equipe de testes na minha equipe de desenvolvimento, eles podem fazer o teste da API aqui.

[11:24] Descendo, tem o botão execute. Clicando, ele dispara e mostra o que foi devolvido. É bem interessante, dá para navegar pela documentação.

[11:48] Lembram do DELETE? Se eu testar, ele vai pedir para passar o id. A url para excluir um tópico é restrita, precisa ter autenticação. Ele também detectou a AutenticacaoController, que é nosso controller para se autenticar. Posso pegar o token e tentar novamente. Porém, onde eu mando os cabeçalhos? Não tem um lugar para isso, porque é um detalhe de segurança da nossa API e não ensinamos isso para o Swagger. Ele não sabe como se autenticar. Esse vai ser o assunto do próximo vídeo.

No último vídeo, vimos como configurar o **SpringFox Swagger**, para gerar a documentação, de maneira automatizada, da nossa API. Nas configurações, foi necessário adicionar o seguinte trecho de código:

ignoredParameterTypes(Usuario.class)

Por que essa configuração foi necessária?

Para ignorar nos *endpoints* os parâmetros relacionados à classe Usuario

Como nossa classe Usuario possui atributos relacionados ao login, senha e perfis de acesso, não é recomendado que essas informações sejam expostas na documentação do Swagger.

@Configuration

public class SwaggerConfigurations {

@Bean

public Docket api() {

return new Docket(DocumentationType.SWAGGER\_2)

.select()

.apis(RequestHandlerSelectors.basePackage("br.com.alura.forum"))

.paths(PathSelectors.ant("/\*\*"))

.build()

.ignoredParameterTypes(Usuario.class)

.globalOperationParameters(

Arrays.asList(

new ParameterBuilder()

.name("Authorization")

.description("Header para Token JWT")

.modelRef(new ModelRef("string"))

.parameterType("header")

.required(false)

.build()));

}

}

Caso você não queira digitar o código da classe SecurityConfigurations mostrado no vídeo anterior, pode copiar daqui:

@Override

public void configure(WebSecurity web) throws Exception {

web.ignoring()

.antMatchers("/\*\*.html", "/v2/api-docs", "/webjars/\*\*", "/configuration/\*\*", "/swagger-resources/\*\*");

}

Nesta aula, aprendemos que:

* Para documentar a nossa API Rest, podemos utilizar o Swagger, com o módulo **SpringFox Swagger**;
* Para utilizar o **SpringFox Swagger** na API, devemos adicionar suas dependências no arquivo **pom.xml**;
* Para habilitar o Swagger na API, devemos adicionar a anotação @EnableSwagger2 na classe ForumApplication;
* As configurações do Swagger devem ser feitas criando-se uma classe chamada SwaggerConfigurations e adicionando nela a anotação @Configuration;
* Para configurar quais *endpoints* e pacotes da API o Swagger deve gerar a documentação, devemos criar um método anotado com @Bean, que devolve um objeto do tipo Docket;
* Para acessar a documentação da API, devemos entrar no endereço <http://localhost:8080/swagger-ui.html>;
* Para liberar acesso ao Swagger no Spring Security, devemos chamar o seguinte método web.ignoring().antMatchers("/\*\*.html", "/v2/api-docs", "/webjars/\*\*", "/configuration/\*\*", "/swagger-resources/\*\*"), dentro do método void configure(WebSecurity web), que está na classe SecurityConfigurations.