## Building a jBPM6 application with SpringBoot

**Details**

|  |  |
| --- | --- |
|  |  |

In this article we will show how you can use jBPM 6 runtime engine with a SpringBoot microservices application

In order to integrate jBPM 6 with Spring you can use two different approaches:

* Use a ****Self Managed Process Engine**** which requires just a single RuntimeManager instance. This can be carried out with the RuntimeManager API
* Use a ****Shared Task Service****: this allows using multiple Runtime Managers and can be carried out using the jBPM Services API

We will use the second approach which requires a bit more of configuration however it allows jBPM assets to be added and removed dynamically without restarting the application. One example you can use to get started quickly is available at: [https://github.com/mswiderski/jbpm-examples/tree/master/spring-boot-jbpm](https://github.com/mswiderski/jbpm-examples/tree/master/spring-boot-jbpm" \t "/home/ryan/Documents\\x/_blank)

Let's see the most significant configuration assets before compiling and running the application.

The first item you will need is an implementation for the ****IdentityProvider**** interface:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | **package** org.jbpm.spring;    **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;    **import** org.kie.internal.identity.IdentityProvider;  **import** org.springframework.security.core.Authentication;  **import** org.springframework.security.core.GrantedAuthority;  **import** org.springframework.security.core.context.SecurityContextHolder;    **public** **class** SpringSecurityIdentityProvider **implements** IdentityProvider {    **public** String getName() {            Authentication auth = SecurityContextHolder.getContext().getAuthentication();  **if** (auth != **null** && auth.isAuthenticated()) {  **return** auth.getName();          }  **return** "system";      }    **public** List<String> getRoles() {          Authentication auth = SecurityContextHolder.getContext().getAuthentication();  **if** (auth != **null** && auth.isAuthenticated()) {              List<String> roles = **new** ArrayList<String>();    **for** (GrantedAuthority ga : auth.getAuthorities()) {                  roles.add(ga.getAuthority());              }    **return** roles;          }    **return** Collections.emptyList();      }    **public** **boolean** hasRole(String role) {  **return** **false**;      }    } |

As you can see, the ****SpringSecurityIdentityProvider**** implements a generic IdentityProvider. The actual Security Context is defined in the ****config/security-context.xml**** file:

|  |  |
| --- | --- |
| 1  2  3  4  5 | <**authentication-manager**>    <**authentication-provider**>      <**user-service** properties="classpath:/roles.properties"/>    </**authentication-provider**>  </**authentication-manager**> |

Let's dig a bit more into the configuration. Next thing is to configure the ****Transaction Manager**** as Spring Bean:

|  |  |
| --- | --- |
| 1  2 | <**bean** id="transactionManager"    class="org.springframework.transaction.jta.JtaTransactionManager" depends-on="bitronixTransactionManager,datasource" /> |

Next it's the turn of ****JPA and Persistence****:

|  |  |
| --- | --- |
| 1  2  3 | <**bean** id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" depends-on="transactionManager">      <**property** name="persistenceXmlLocation" value="classpath:/META-INF/jbpm-persistence.xml" />  </**bean**> |

Finally, it's the turn of ****Runtime Manager Factory**** and ****Jbpm Services****:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | <**bean** id="runtimeManagerFactory" class="org.kie.spring.manager.SpringRuntimeManagerFactoryImpl">    <**property** name="transactionManager" ref="transactionManager"/>    <**property** name="userGroupCallback" ref="userGroupCallback"/>  </**bean**>    <**bean** id="definitionService" class="org.jbpm.kie.services.impl.bpmn2.BPMN2DataServiceImpl"/>    <**bean** id="taskService" class="org.kie.spring.factorybeans.TaskServiceFactoryBean" destroy-method="close">    <**property** name="entityManagerFactory" ref="entityManagerFactory"/>    <**property** name="transactionManager" ref="transactionManager"/>    <**property** name="userGroupCallback" ref="userGroupCallback"/>    <**property** name="listeners">      <**list**>        <**bean** class="org.jbpm.services.task.audit.JPATaskLifeCycleEventListener">          <**constructor-arg** value="true"/>        </**bean**>      </**list**>    </**property**>  </**bean**>    <**bean** id="transactionCmdService" class="org.jbpm.shared.services.impl.TransactionalCommandService">    <**constructor-arg** name="emf" ref="entityManagerFactory"></**constructor-arg**>  </**bean**>    <**bean** id="runtimeDataService" class="org.jbpm.kie.services.impl.RuntimeDataServiceImpl">    <**property** name="commandService" ref="transactionCmdService"/>    <**property** name="identityProvider" ref="identityProvider"/>    <**property** name="taskService" ref="taskService"/>  </**bean**>    <**bean** id="deploymentService" class="org.jbpm.kie.services.impl.KModuleDeploymentService" depends-on="entityManagerFactory" init-method="onInit">    <**property** name="bpmn2Service" ref="definitionService"/>    <**property** name="emf" ref="entityManagerFactory"/>    <**property** name="managerFactory" ref="runtimeManagerFactory"/>    <**property** name="identityProvider" ref="identityProvider"/>    <**property** name="runtimeDataService" ref="runtimeDataService"/>  </**bean**> |

By including the appropriate dependencies such as jbpm-services-api, jbpm-kie-services, and kie-spring api you will be able to @Autowire jbpm Objects in SpringBoot:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <**dependency**>      <**groupId**>org.jbpm</**groupId**>      <**artifactId**>jbpm-services-api</**artifactId**>      <**version**>${jbpm.version}</**version**>    </**dependency**>    <**dependency**>      <**groupId**>org.jbpm</**groupId**>      <**artifactId**>jbpm-kie-services</**artifactId**>      <**version**>${jbpm.version}</**version**>    </**dependency**>    <**dependency**>      <**groupId**>org.kie</**groupId**>      <**artifactId**>kie-spring</**artifactId**>      <**version**>${jbpm.version}</**version**>    </**dependency**> |

You can check the full ****pom.xml**** required to compile your project at: [https://github.com/mswiderski/jbpm-examples/blob/master/spring-boot-jbpm/pom.xml](https://github.com/mswiderski/jbpm-examples/blob/master/spring-boot-jbpm/pom.xml" \t "/home/ryan/Documents\\x/_blank)

Now, as an example, consider the following ****ProcessInstanceController**** which will let you show, signal or abort existing Processes using the REST Api:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | @RestController  @RequestMapping("/processinstance")  **public** **class** ProcessInstanceController {        @Autowired  **private** RuntimeDataService runtimeDataService;        @Autowired  **private** ProcessService processService;        @RequestMapping(value = "/", method = RequestMethod.GET)  **public** Collection<ProcessInstanceDesc> getProcessInstances() {            Collection<ProcessInstanceDesc> processInstances = runtimeDataService.getProcessInstances(**new** QueryContext(0, 100, "status", **true**));    **return** processInstances;        }        @RequestMapping(value = "/show", method = RequestMethod.GET)  **public** ProcessInstanceDesc getProcessInstance(@RequestParam String id) {  **long** processInstanceId = Long.parseLong(id);          ProcessInstanceDesc processInstance = runtimeDataService.getProcessInstanceById(processInstanceId);    **return** processInstance;      }        @RequestMapping(value = "/abort", method = RequestMethod.POST)  **public** String abortProcessInstance(@RequestParam String id) {            processService.abortProcessInstance(Long.parseLong(id));    **return** "Instance (" + id + ") aborted successfully";      }        @RequestMapping(value = "/signal", method = RequestMethod.POST)  **public** String signalProcessInstance(@RequestParam String id, @RequestParam String signal,              @RequestParam String data) {            processService.signalProcessInstance(Long.parseLong(id), signal, data);    **return** "Signal sent to instance (" + id + ") successfully";      }  } |

On the other hand, in order to start new Process Instances you can use the following ****ProcessDefController**** and the "****/new"**** Rest API:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40 | @RestController  @RequestMapping("/processdef")  **public** **class** ProcessDefController {        @Autowired  **private** RuntimeDataService runtimeDataService;        @Autowired  **private** ProcessService processService;        @Autowired  **private** DefinitionService definitionService;        @RequestMapping(value = "/", method = RequestMethod.GET)  **public** Collection<ProcessDefinition> getProcessDef() {            Collection<ProcessDefinition> processDefinitions = runtimeDataService.getProcesses(**new** QueryContext(0, 100));    **return** processDefinitions;        }        @RequestMapping(value = "/show", method = RequestMethod.GET)  **public** ProcessDefinition getProcessDefinition(@RequestParam String deployment, @RequestParam String id) {            ProcessDefinition definition = runtimeDataService.getProcessesByDeploymentIdProcessId(deployment, id);    **return** definition;      }        @RequestMapping(value = "/new", method = RequestMethod.POST)  **public** Long newProcessInstance(@RequestParam String deploymentId, @RequestParam String processId,              @RequestParam Map<String,String> allRequestParams) {    **long** processInstanceId = processService.startProcess(deploymentId, processId, **new** HashMap<String, Object>(allRequestParams));    **return** processInstanceId;        }  } |

Let's test it! Before running our SpringBoot Rest application, let's make available in our local git repository a jbpm artifact. Download the following sample project which contains a simple Business Rule and a jBPM project: [https://github.com/jesuino/hello-kie-server](https://github.com/jesuino/hello-kie-server" \t "/home/ryan/Documents\\x/_blank)

Install the jbpm artifact with Maven:

|  |  |
| --- | --- |
| 1  2 | $ cd hello-kie-server  $ mvn clean install |

As a result, you should have available in your Maven repository the following Group Artifact Version combination:

|  |  |
| --- | --- |
| 1  2  3 | <**group-id**>org.mastertheboss.kieserver</**group-id**>  <**artifact-id**>hello-kie-server</**artifact-id**>  <**version**>1.0</**version**> |

Now let's get back to the jbpm-springboot application. Compile it and install it with:

|  |  |
| --- | --- |
| 1  2 | $ cd spring-boot-jbpm  $ mvn clean install |

Execute the SpringBoot application, passing as argument the GAV combination :

|  |  |
| --- | --- |
| 1 | java -jar target/spring-boot-jbpm-0.0.1-SNAPSHOT.jar org.mastertheboss.kieserver hello-kie-server 1.0 |

The embedded Tomcat Server will start on the default port 8080.

If you take a look at the ****Application**** class, you will see that the startup arguments are used to Deploy the Process definition as KModuleDeploymentUnit:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | @Configuration  @ComponentScan  @EnableAutoConfiguration(exclude=HibernateJpaAutoConfiguration.**class**)  @ImportResource(value= {"classpath:config/jee-tx-context.xml",          "classpath:config/jpa-context.xml", "classpath:config/jbpm-context.xml", "classpath:config/security-context.xml",})  **public** **class** Application {    **public** **static** **void** main(String[] args) {          ConfigurableApplicationContext ctx = SpringApplication.run(Application.**class**, args);  **if** (args.length > 1) {    **try** {                  System.out.println("Params available trying to deploy " + args);                  DeploymentService deploymentService = (DeploymentService) ctx.getBean("deploymentService");                    KModuleDeploymentUnit unit = **new** KModuleDeploymentUnit(args[0], args[1], args[2]);                  deploymentService.deploy(unit);              } **catch** (Throwable e) {                  System.out.println("Error when deploying = " + e.getMessage());              }          }      }  } |

Let's test our application! first of all, we'll see if the sample hello-kie-server is found from the local Repository. We will use the ProcessDefController and its "/show" Api for this purpose:

|  |  |
| --- | --- |
| 1 | $ curl -X GET  -u 'john:john1' "http://localhost:8080/processdef/show?id=hello&deployment=org.mastertheboss.kieserver:hello-kie-server:1.0" |

Notice we had to pass the login credentials (found in the roles.properties file) and some @RequestParam attribute to identify the ProcessDefinition

Here's the output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | {        "id": "hello",      "name": "hello",      "version": "1.0",      "packageName": "org.jbpm",      "type": "RuleFlow",      "knowledgeType": "PROCESS",      "namespace": "org.jbpm",      "originalPath": null,      "deploymentId": "org.mastertheboss.kieserver:hello-kie-server:1.0",      "encodedProcessSource":     . . . .      } |

Now let's start one process instance, using the again the ProcessDefController and its "/new" Api:

|  |  |
| --- | --- |
| 1 | curl -X POST  -u 'john:john1' --data 'deploymentId=org.mastertheboss.kieserver:hello-kie-server:1.0&processId=hello&name=Frank' http://localhost:8080/processdef/new |

Finally, let's check through the ProcessInstanceController and its "/processinstance" request path, the list of processes running:

|  |  |
| --- | --- |
| 1 | curl -X GET  -u 'john:john1' "http://localhost:8080/processinstance/" |

Here's the output:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | [        {          "id": 1,          "processId": "hello",          "processName": "hello",          "processVersion": "1.0",          "state": 2,          "deploymentId": "org.mastertheboss.kieserver:hello-kie-server:1.0",          "initiator": "john",          "dataTimeStamp": 1494088784933,          "processInstanceDescription": "hello",          "activeTasks": null      }    ] |

For the sake of completeness, we'll mention that you can check individual process instance state, through the "/show" method, passing the Process Id as argument:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | curl -X GET -u 'john:john1' http://localhost:8080/processinstance/show?id=1    {        "id": 1,      "processId": "hello",      "processName": "hello",      "processVersion": "1.0",      "state": 2,      "deploymentId": "org.mastertheboss.kieserver:hello-kie-server:1.0",      "initiator": "john",      "dataTimeStamp": 1494088784933,      "processInstanceDescription": "hello",      "activeTasks": [ ]    } |

That's all! enjoy ****SpringBoot**** and ****jBPM 6****!