# Runtime Governance: Quick Start Guide

# Chapter 1. Introduction

This guide provides an introduction to the quickstarts that are distributed with the Overlord Runtime Governance project.

# Chapter 2. Policy Enforcement

This example, located in the `samples/policy` folder, demonstrates two approaches that can be used to provide "policy enforcement". This example makes uses of the example Switchyard application located in the `samples/ordermgmt` folder.

## 2.1. Synchronous Enforcement

The first approach shows how a business policy can be implemented in a synchronous (or inline) manner, where the decision is taken immediate, and can therefore be used to influence the current business transaction. The benefit of this approach is that it can ensure only valid transactions are permitted, as decisions can be immediately enforced, however the disadvantage is the potential performance impact this may have on the business transaction.

This example will show how:

- activity event analysis, using the Activity Validator mechanism, to implement the business policy

### 2.1.1. The Policy

The runtime governance infrastructure analyses the activity events generated by an executing business transaction using one or more Activity Validators. By default, Activity Validators are not invoked from within the SwitchYard environment. The specific SwitchYard applications need to be configured to include an interceptor that will invoke the validation. In the Order Management quickstart, this is achieved using the class `org.overlord.rtgov.quickstarts.demos.orders.interceptors.ExchangeValidator`. This class is derived from an abstract base class that provides most of the required functionality for converting an Exchange message into an activity event. For example,

```
@Named("ExchangeValidator")
public class ExchangeValidator extends AbstractExchangeValidator implements
 ExchangeInterceptor {

    @Override
    public void before(String target, Exchange exchange) throws
 HandlerException {
        if (exchange.getPhase() == ExchangePhase.IN) {
            handleExchange(exchange);
        }
    }

    @Override
    public void after(String target, Exchange exchange) throws
 HandlerException {
        if (exchange.getPhase() == ExchangePhase.OUT) {
            handleExchange(exchange);
```

```
            }
        }

        @Override
        public List<String> getTargets() {
            return Arrays.asList(PROVIDER);
        }
    }
```

The following Activity Validator configuration is deployed in the environment responsible for executing the business transaction, and gets registered with the Activity Collector mechanism:

```
[{
    "name" : "RestrictUsage",
    "version" : "1",
    "predicate" : {
        "@class" : "org.overlord.rtgov.ep.mvel.MVELPredicate",
        "expression" : "event instanceof
 org.overlord.rtgov.activity.model.soa.RequestReceived && event.serviceType
 == \"{urn:switchyard-quickstart-demo:orders:0.1.0}OrderService\""
    },
    "eventProcessor" : {
        "@class" : "org.overlord.rtgov.ep.mvel.MVELEventProcessor",
        "script" : "VerifyLastUsage.mvel",
        "services" : {
            "CacheManager" : {
                "@class" :
 "org.overlord.rtgov.common.infinispan.service.InfinispanCacheManager"
            }
        }
    }
}]
```

This Activity Validator receives activity events generated from the executing environment and applies the optional predicate to determine if they are of interest to the defined event processor. In this case, the predicate is checking for received requests for the `OrderService` service.

For events that pass this predicate, they are submitted to the *business policy*, defined using the MVEL script `VerifyLastUsage.mvel`, which is:

```
String customer=event.properties.get("customer");

if (customer == null) {
 return;
}

cm = epc.getService("CacheManager");
```

```
// Attempt to lock the entry
if (!cm.lock("Principals", customer)) {
 epc.handle(new java.lang.RuntimeException("Unable to lock entry for
 principal '"+customer+"'"));

 return;
}

// Access the cache of principals
principals = cm.getCache("Principals");

principal = principals.get(customer);

if (principal == null) {
 principal = new java.util.HashMap();
}

java.util.Date current=principal.get(event.serviceType+"-lastaccess");
java.util.Date now=new java.util.Date();

if (current != null && (now.getTime()-current.getTime()) < 2000) {
 epc.handle(new java.lang.RuntimeException("Customer '"+customer+"' cannot
 perform more than one request every 2 seconds"));

 return;
}

principal.put(event.serviceType+"-lastaccess", now);
principals.put(customer, principal);

epc.logDebug("Updated principal '"+customer+"': "+principals.get(customer));
```

This script uses the *CacheManager* service, configured within the EventProcessor component, to obtain a cache called "Principals". This cache is used to store information about Principals as a map of properties. The implementation uses Infinispan, to enable the cache to be shared between other applications, as well as in a distributed/cluster environment (based on the infinispan configuration).

If a policy violation is detected, the script returns an exception using the *handle()* method on the EventProcessor context. This results in the exception being thrown back to the execution environment, interrupting the execution of the business transaction.

## 2.1.2. Installation

To install the example, the first step is to start the Switchyard server using the following command from the `bin` folder:

```
./standalone.sh -c standalone-full.xml
```

The next step is to install the example Switchyard application, achieved by running the following command from the `${rtgov}/samples/ordermgmt` folder:

```
mvn jboss-as:deploy
```

Then change to the `${rtgov}/samples/policy/sync` folder and run the same command again.

## 2.1.3. Running the Example

To demonstrate the synchronous policy, we will send the following message twice in less than 2 seconds, to the example Switchyard application at the following URL: *http://localhost:8080/demo-orders/OrderService*

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <orders:submitOrder xmlns:orders="urn:switchyard-quickstart-
demo:orders:1.0">
            <order>
                <orderId>1</orderId>
                <itemId>BUTTER</itemId>
                <quantity>100</quantity>
              <customer>Fred</customer>
            </order>
        </orders:submitOrder>
    </soap:Body>
</soap:Envelope>
```

The messages can be sent using an appropriate SOAP client (e.g. SOAP-UI) or by running the test client available with the Switchyard application, by running the following command from the `${rtgov}/samples/ordermgmt/app` folder:

```
mvn exec:java -Dreq=order1 -Dcount=2
```

If the two requests are received within two seconds of each other, this will result in the following response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Server</faultcode>
         <faultstring>org.switchyard.exception.SwitchYardException: Customer
 'Fred' cannot perform more than one request every 2 seconds</faultstring>
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

## 2.1.4. Summary

This quickstart example demonstrates how a policy enforcement mechanism can be provided using the Activity Validator mechanism, to immediately evaluate the business policy and (if appropriate) block the business transaction.

# 2.2. Asynchronous Enforcement

The second approach shows how a business policy can be implemented in an asynchronous (or out-of-band) manner, where the decision is taken after the fact, and can therefore only be used to influence future business transactions. The benefit of this approach is that the decision making process does not have to occur immediately and therefore avoids potentially impacting the performance of the business transaction. The disadvantage is that it does not permit any decision that is made to be enforced immediately.

This example will show how:

- activity event analysis, using the Event Processor Network mechanism, can be used to implement business policies

- results from the business policies can be cached for reference by other applications

- platform specific interceptors can reference the results to impact the behavior of the business transaction (e.g. prevent suspended customers purchasing further items)

## 2.2.1. The Policy

There are three components that comprise *the policy* within this example.

### 2.2.1.1. Event analysis

The runtime governance infrastructure analyses the activity events generated by an executing business transaction using one or more Event Processor Networks (or EPN).

A standard EPN is deployed within the infrastructure to isolate the SOA events (e.g. request/ responses being sent or received). This quickstart deploys another EPN that subscribes to the events produced by the standard EPN:

```
{
  "name" : "AssessCreditPolicyEPN",
  "version" : "1",
  "subscriptions" : [ {
    "nodeName" : "AssessCredit",
    "subject" : "SOAEvents"
  } ],
  "nodes" : [
    {
      "name" : "AssessCredit",
```

```
      "sourceNodes" : [ ],
      "destinationSubjects" : [ ],
      "maxRetries" : 3,
      "retryInterval" : 0,
      "predicate" : {
        "@class" : "org.overlord.rtgov.ep.mvel.MVELPredicate",
        "expression" : "event.serviceProvider && !event.request
 && event.serviceType == \"{urn:switchyard-quickstart-
demo:orders:0.1.0}OrderService\""
      },
      "eventProcessor" : {
        "@class" : "org.overlord.rtgov.ep.mvel.MVELEventProcessor",
        "script" : "AssessCredit.mvel",
        "services" : {
          "CacheManager" : {
            "@class" :
 "org.overlord.rtgov.common.infinispan.service.InfinispanCacheManager"
          }
        }
      }
    }
  ]
}
```

This EPN subscribes to the published SOA events and applies the predicate which ensures that only events from a service provider interface, that are responses and are associated with the `OrderService` service, will be processed. Events that pass this predicate are then submitted to the *business policy* (defined in the MVEL script *AssessCredit.mvel*), which is:

```
String customer=event.properties.get("customer");

if (customer == null) {
 return;
}

cm = epc.getService("CacheManager");

// Attempt to lock the entry
if (!cm.lock("Principals", customer)) {
 epc.handle(new Exception("Unable to lock entry for principal '"+customer
+"'"));

 return;
}

// Access the cache of principals
principals = cm.getCache("Principals");

principal = principals.get(customer);
```

```
if (principal == null) {
 principal = new java.util.HashMap();
}

int current=principal.get("exposure");

if (current == null) {
 current = 0;
}

if (event.operation == "submitOrder") {

 double total=event.properties.get("total");

 int newtotal=current+total;

 if (newtotal > 150 && current <= 150) {
  principal.put("suspended", Boolean.TRUE);
 }

 principal.put("exposure", newtotal);

} else if (event.operation == "makePayment") {

 double amount=event.properties.get("amount");

 int newamount=current-amount;

 if (newamount <= 150 && current > 150) {
  principal.put("suspended", Boolean.FALSE);
 }

 principal.put("exposure", newamount);
}

principals.put(customer, principal);

epc.logDebug("Updated principal '"+customer+"': "+principals.get(customer));
```

This script uses the *CacheManager* service, configured within the EPN node, to obtain a cache called "Principals". This cache is used to store information about Principals as a map of properties. The implementation uses Infinispan, to enable the cache to be shared between other applications, as well as in a distributed/cluster environment (based on the infinispan configuration).

## 2.2.1.2. Result management

As mentioned in the previous section, the results derived from the previous policy are stored in an Infinispan implemented cache called "Principals". To make this information available to runtime

governance clients, we use the Active Collection mechanism - more specifically we define an Active Collection, as part of the standard installation, that wraps the Infinispan cache.

The configuration of the Active Collection Source is:

```
[
  {
    ......
  },{
    "@class" :
 "org.overlord.rtgov.active.collection.ActiveCollectionSource",
    "name" : "Principals",
    "type" : "Map",
    "lazy" : true,
    "visibility" : "Private",
    "factory" : {
        "@class" :
 "org.overlord.rtgov.active.collection.infinispan.InfinispanActiveCollectionFactory",
        "cache" : "Principals"
    }
  }
]
```

The *visibility* is marked as **private** to ensure that exposure information regarding customers is not publicly available via the Active Collection REST API.

## 2.2.1.3. The enforcer

The enforcement is provided by a specific Switchyard exchange interceptor implementation (PolicyEnforcer) that is included with the order management application. The main part of this interceptor is:

```
    public void before(String call, Exchange exch) throws HandlerException {
        ....

        if (_principals != null) {
            org.switchyard.Message mesg=exch.getMessage();

            if (mesg == null) {
                LOG.severe("Could not obtain message for phase ("+phase+")
 and exchange: "+exch);
                return;
            }

            org.switchyard.Context context=exch.getContext();

            Property p=context.getProperty(Exchange.CONTENT_TYPE,
 org.switchyard.Scope.MESSAGE);
```

```
           if (LOG.isLoggable(Level.FINER)) {
               LOG.finer("Content type="+(p==null?null:p.getValue()));
           }

           if (p != null && p.getValue().toString().equals(

  "java:org.overlord.rtgov.quickstarts.demos.orders.Order")) {

               String customer=getCustomer(mesg);

               if (customer != null) {
                   if (_principals.containsKey(customer)) {

                       @SuppressWarnings("unchecked")
                       java.util.Map<String,java.io.Serializable> props=
                               (java.util.Map<String,java.io.Serializable>)
                                   _principals.get(customer);

                       // Check if customer is suspended
                       if (props.containsKey("suspended")
                               &&
  props.get("suspended").equals(Boolean.TRUE)) {
                           throw new HandlerException("Customer '"+customer
                                       +"' has been suspended");
                       }
                   }

                   if (LOG.isLoggable(Level.FINE)) {
                       LOG.fine("*********** Policy Enforcer: customer '"
                               +customer+"' has not been suspended");
                       LOG.fine("*********** Principal:
  "+_principals.get(customer));
                   }
               } else {
                   LOG.warning("Unable to find customer name");
               }
           }
       }
   }
```

The variable *_principals* refers to an Active Map used to maintain information about the principal (i.e. the customer in this case). This information is updated using the policy rule defined in the previous section.

## 2.2.2. Installation

To install the example, the first step is to start the Switchyard server using the following command from the `bin` folder:

```
    ./standalone.sh -c standalone-full.xml
```

The next step is to install the example Switchyard application, achieved by running the following command from the `${rtgov}/samples/ordermgmt` folder:

```
    mvn jboss-as:deploy
```

Then change to the `${rtgov}/samples/policy/async` folder and run the same command again.

## 2.2.3. Running the Example

To demonstrate the asynchronous policy enforcement, we will send the following message to the example Switchyard application at the following URL: *http://localhost:8080/demo-orders/OrderService*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <orders:submitOrder xmlns:orders="urn:switchyard-quickstart-
demo:orders:1.0">
            <order>
                <orderId>1</orderId>
                <itemId>BUTTER</itemId>
                <quantity>100</quantity>
              <customer>Fred</customer>
            </order>
        </orders:submitOrder>
    </soap:Body>
</soap:Envelope>
```

The message can be sent using an appropriate SOAP client (e.g. SOAP-UI) or by running the test client available with the Switchyard application, by running the following command from the `${rtgov}/samples/ordermgmt/app` folder:

```
    mvn exec:java -Dreq=order1
```

This will result in the following response, indicating that the purchase was successful, as well as identifying the total cost of the purchase (i.e. 125).

```xml
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/">
    <SOAP-ENV:Header/>
    <SOAP-ENV:Body>
        <orders:submitOrderResponse xmlns:orders="urn:switchyard-quickstart-
demo:orders:1.0">
            <orderAck>
                <orderId>1</orderId>
```

```
            <accepted>true</accepted>
            <status>Order Accepted</status>
            <customer>Fred</customer>
            <total>125.0</total>
        </orderAck>
    </orders:submitOrderResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

You may recall from the overview above that if the customer's debt exceeds the threshold of 150 then the customer would be suspended. Therefore if the same request is issued again, resulting in another total of 125, then the overall exposure regarding this customer is now 250.

If we then attempt to issue the same request a third time, this time we will receive a SOAP fault from the server. This is due to the fact that the PolicyEnforcer has intercepted the request, and detected that the customer is now suspended.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/
envelope/">
   <SOAP-ENV:Header/>
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Server</faultcode>
         <faultstring>Customer 'Fred' has been suspended</faultstring>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

If we now send a "makePayment" request as follows to the same URL:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:urn="urn:switchyard-quickstart-demo:orders:1.0">
   <soapenv:Header/>
   <soapenv:Body>
      <urn:makePayment>
         <payment>
            <customer>Fred</customer>
            <amount>200</amount>
         </payment>
      </urn:makePayment>
   </soapenv:Body>
</soapenv:Envelope>
```

This can be sent using a suitable SOAP client (e.g. SOAP-UI) or the test client in the order management app:

```
mvn exec:java -Dreq=fredpay
```

This will result in the customer being unsuspended, as it removes 200 from their current exposure (leaving 50). To confirm this, try sending the "submitOrder" request again.

## 2.2.4. Summary

This quickstart example demonstrates how a policy enforcement mechanism can be provided using a combination of the Runtime Governance infrastructure and platform specific interceptors.

This particular example uses an asynchronous approach to evaluate the business policies, only enforcing the policy based on a summary result from the decision making process. The benefit of this approach is that it can be more efficient, and reduce the performance impact on the business transaction being policed. The disadvantage is that decisions are made after the fact, so leaves a tiny window of opportunity for invalid transactions to be performed.

# Chapter 3. SLA

## 3.1. Monitor

This example, located in the `samples/sla/monitor` folder, demonstrates an approach to provide "Service Level Agreement" monitoring. This example makes uses of the example Switchyard application located in the `samples/ordermgmt` folder.

### 3.1.1. Overview

This example will show how:

- activity event analysis, using the Event Processor Network mechanism, can be used to implement Service Level Agreements

  - uses the Complex Event Processing (CEP) based event processor (using Drools Fusion)

- impending or actual SLA violations can be reported for the attention of end users, via

  - JMX notifications

  - REST service

- to build a custom application to access the analysis results

This example shows a simple Service Level Agreement that checks whether a service response time exceeds expected levels. The CEP rule detects whether a situation of interest has occurred, and if so, creates a `org.overlord.rtgov.analytics.situation.Situation` object and initializes it with the appropriate description/severity information, before forwarding it back into the EPN. This results in the "Situation" object being published as a notification on the "Situations" subject.

The CEP rule is:

```
import org.overlord.rtgov.analytics.service.ResponseTime
import org.overlord.rtgov.analytics.situation.Situation

global org.overlord.rtgov.ep.EPContext epc

declare ResponseTime
    @role( event )
end

rule "check for SLA violations"
when
    $rt : ResponseTime() from entry-point "ServiceResponseTimes"
then
```

```
if ($rt.getAverage() > 200) {
 epc.logError("\r\n\r\n**** RESPONSE TIME "+$rt.getAverage()+"ms EXCEEDED
SLA FOR "+$rt.getServiceType()+" ****\r\n");

 Situation situation=new Situation();

 situation.setType("SLA Violation");
 situation.setSubject(Situation.createSubject($rt.getServiceType(),
$rt.getOperation(),
     $rt.getFault()));
 situation.setTimestamp(System.currentTimeMillis());

 situation.getProperties().putAll($rt.getProperties());

 if ($rt.getRequestId() != null) {
  situation.getActivityTypeIds().add($rt.getRequestId());
 }
 if ($rt.getResponseId() != null) {
  situation.getActivityTypeIds().add($rt.getResponseId());
 }

 situation.getContext().addAll($rt.getContext());

 String serviceName=$rt.getServiceType();

 if (serviceName.startsWith("{")) {
  serviceName =
javax.xml.namespace.QName.valueOf(serviceName).getLocalPart();
 }

 if ($rt.getAverage() > 400) {
  situation.setDescription(serviceName+" exceeded maximum response time of
400 ms");
  situation.setSeverity(Situation.Severity.Critical);
 } else if ($rt.getAverage() > 320) {
  situation.setDescription(serviceName+" exceeded response time of 320
ms");
  situation.setSeverity(Situation.Severity.High);
 } else if ($rt.getAverage() > 260) {
  situation.setDescription(serviceName+" exceeded response time of 260
ms");
  situation.setSeverity(Situation.Severity.Medium);
 } else {
  situation.setDescription(serviceName+" exceeded response time of 200
ms");
  situation.setSeverity(Situation.Severity.Low);
 }

 epc.handle(situation);
```

```
    }

  end
```

The "out of the box" active collection configuration is pre-initialized with a collection for the `org.overlord.rtgov.analytics.situation.Situation` objects, subscribing to the "Situations" subject from the Event Processor Network. Therefore any detected SLA violations will automatically be stored in this collection (accessible via a RESTful service), and reported to the associated JMX notifier.

## 3.1.2. Installation

To install the example, the first step is to start the Switchyard server using the following command from the `bin` folder:

```
   ./standalone.sh -c standalone-full.xml
```

The next step is to install the example Switchyard application, achieved by running the following command from the `${rtgov}/samples/ordermgmt` folder:

```
   mvn jboss-as:deploy
```

Then run the same command from the `${rtgov}/samples/sla/epn` and `${rtgov}/samples/sla/monitor` folders.

## 3.1.3. Running the Example

To demonstrate a Service Level Agreement violation, we will send the following message to the example Switchyard application at the following URL: *http://localhost:8080/demo-orders/OrderService*

```xml
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <orders:submitOrder xmlns:orders="urn:switchyard-quickstart-
demo:orders:1.0">
         <order>
            <orderId>3</orderId>
            <itemId>JAM</itemId>
            <quantity>400</quantity>
            <customer>Fred</customer>
         </order>
      </orders:submitOrder>
   </soap:Body>
</soap:Envelope>
```

The message can be sent using an appropriate SOAP client (e.g. SOAP-UI) or by running the test client available with the Switchyard application, by running the following command from the `${rtgov}/samples/ordermgmt/app` folder:

```
mvn exec:java -Dreq=order3
```

The *itemId* of "JAM" causes a delay to be introduced in the service, resulting in a SLA violation being detected. This violation can be viewed using two approaches:

### 3.1.3.1. REST Service

Using a suitable REST client, send the following POST to: *http://localhost:8080/overlord-rtgov/ acm/query* (using content-type of "application/json", username is *admin* and password is *overlord*)

```
{
    "collection" : "Situations"
}
```

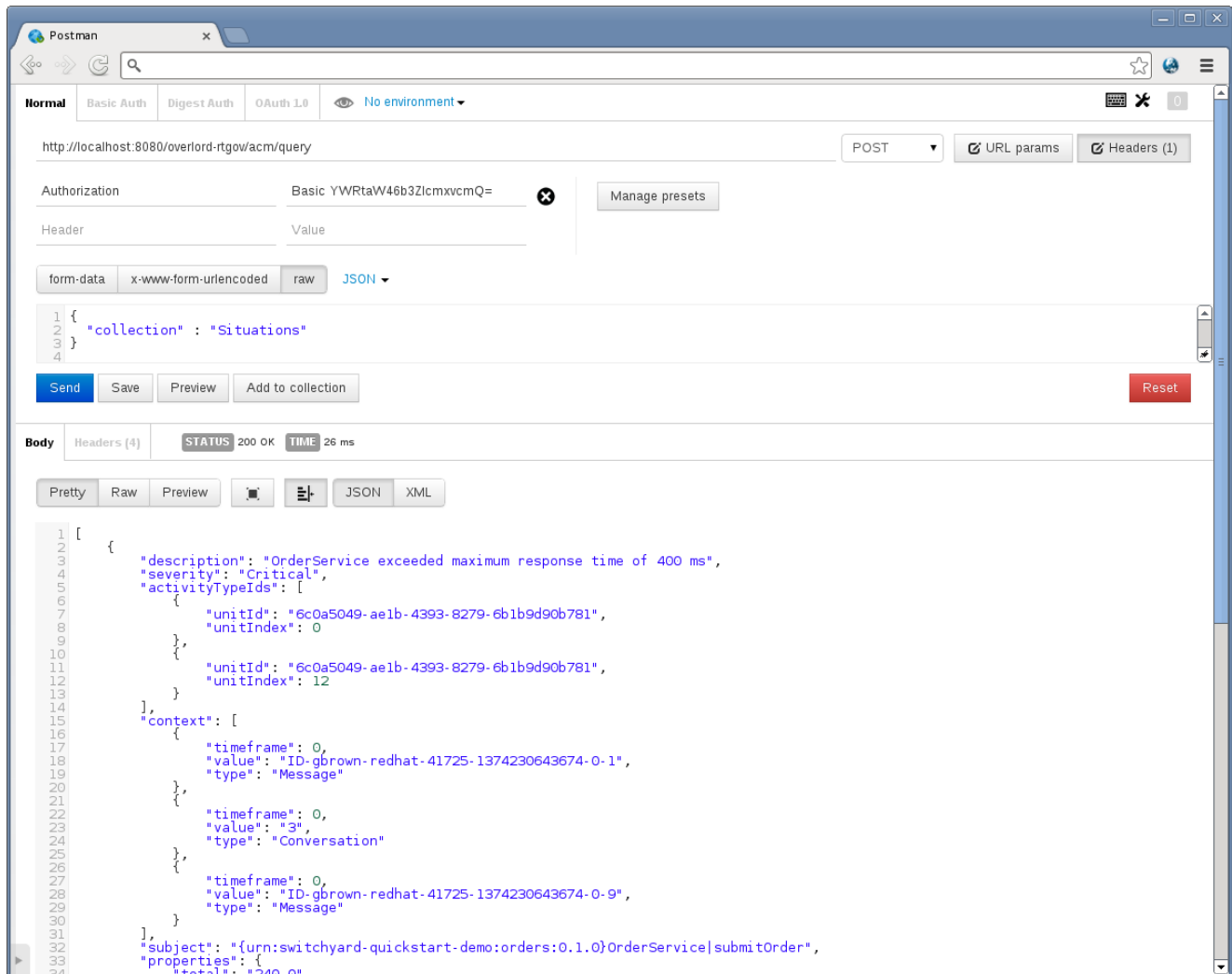This will result in the following response:

**Figure 3.1.**

## 3.1.3.2. JMX Console

The *Situations* active collection source also generates JMX notifications that can be subscribed to using a suitable JMX management application. For example, using JConsole we can view the SLA violation:
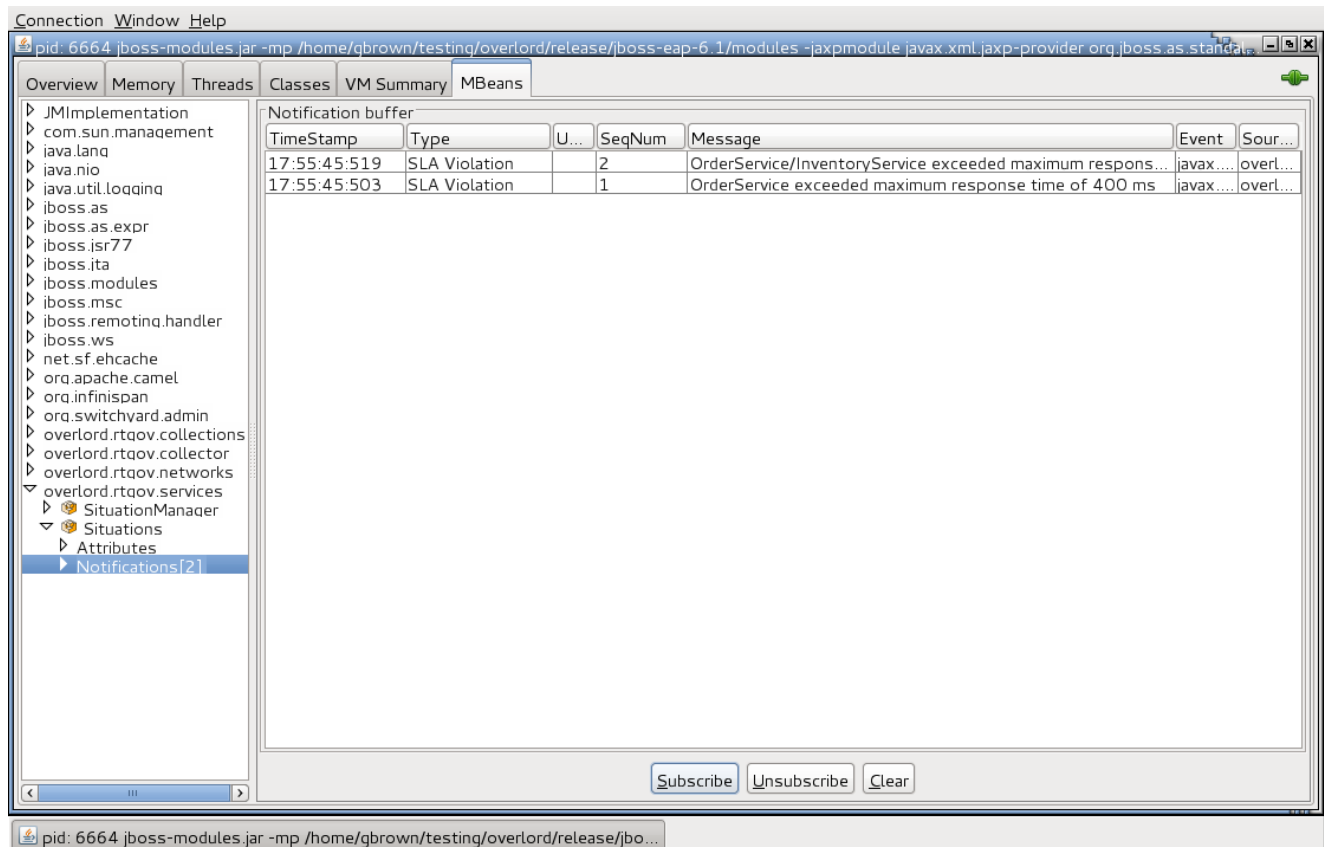
**Figure 3.2.**

## 3.1.3.3. Accessing results within a custom application

As well as having access to the information via REST or JMX, it may also be desirable to have more direct access to the active collection results. This section describes the custom app defined in the `${rtgov}/samples/sla/monitor` folder.

The following code shows how the custom application initializes access to the relevant active collections:

```
@Path("/monitor")
@ApplicationScoped
public class SLAMonitor {

    private static final String SERVICE_RESPONSE_TIMES =
  "ServiceResponseTimes";
    private static final String SITUATIONS = "Situations";

    private static final Logger
  LOG=Logger.getLogger(SLAMonitor.class.getName());

    private ActiveCollectionManager _acmManager=null;
```

```
    private ActiveList _serviceResponseTime=null;
    private ActiveList _situations=null;

    /**
     * This is the default constructor.
     */
    public SLAMonitor() {

        try {
            _acmManager =
ActiveCollectionManagerAccessor.getActiveCollectionManager();

            _serviceResponseTime = (ActiveList)
                    _acmManager.getActiveCollection(SERVICE_RESPONSE_TIMES);

            _situations = (ActiveList)
                    _acmManager.getActiveCollection(SITUATIONS);

        } catch (Exception e) {
            LOG.log(Level.SEVERE, "Failed to initialize active collection
manager", e);
        }

    }
```

Then when the REST request is received (e.g. for SLA violations defined as Situations),

```
    @GET
    @Path("/situations")
    @Produces("application/json")
    public java.util.List<Situation> getSituations() {
        java.util.List<Situation> ret=new java.util.ArrayList<Situation>();

        for (Object obj : _situations) {
            if (obj instanceof Situation) {
                ret.add((Situation)obj);
            }
        }

        return (ret);
    }
```

To see the SLA violations, send a REST GET request to: *http://localhost:8080/slamonitor-monitor/monitor/situations*

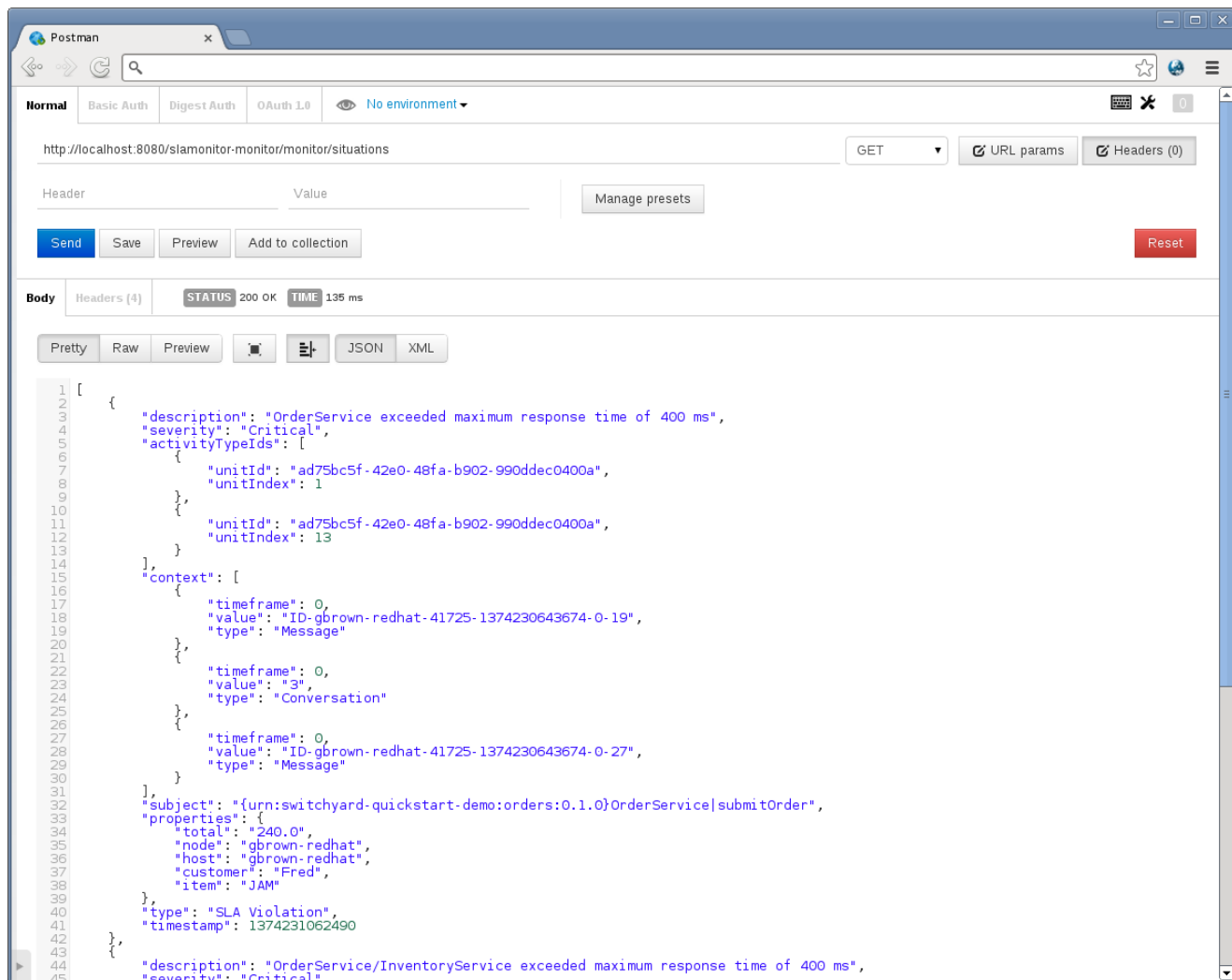This will return the following information:

**Figure 3.3.**

It is also possible to request the list of response time information from the same custom service, using the URL: *http://localhost:8080/slamonitor-monitor/monitor/responseTimes? operation=submitOrder*
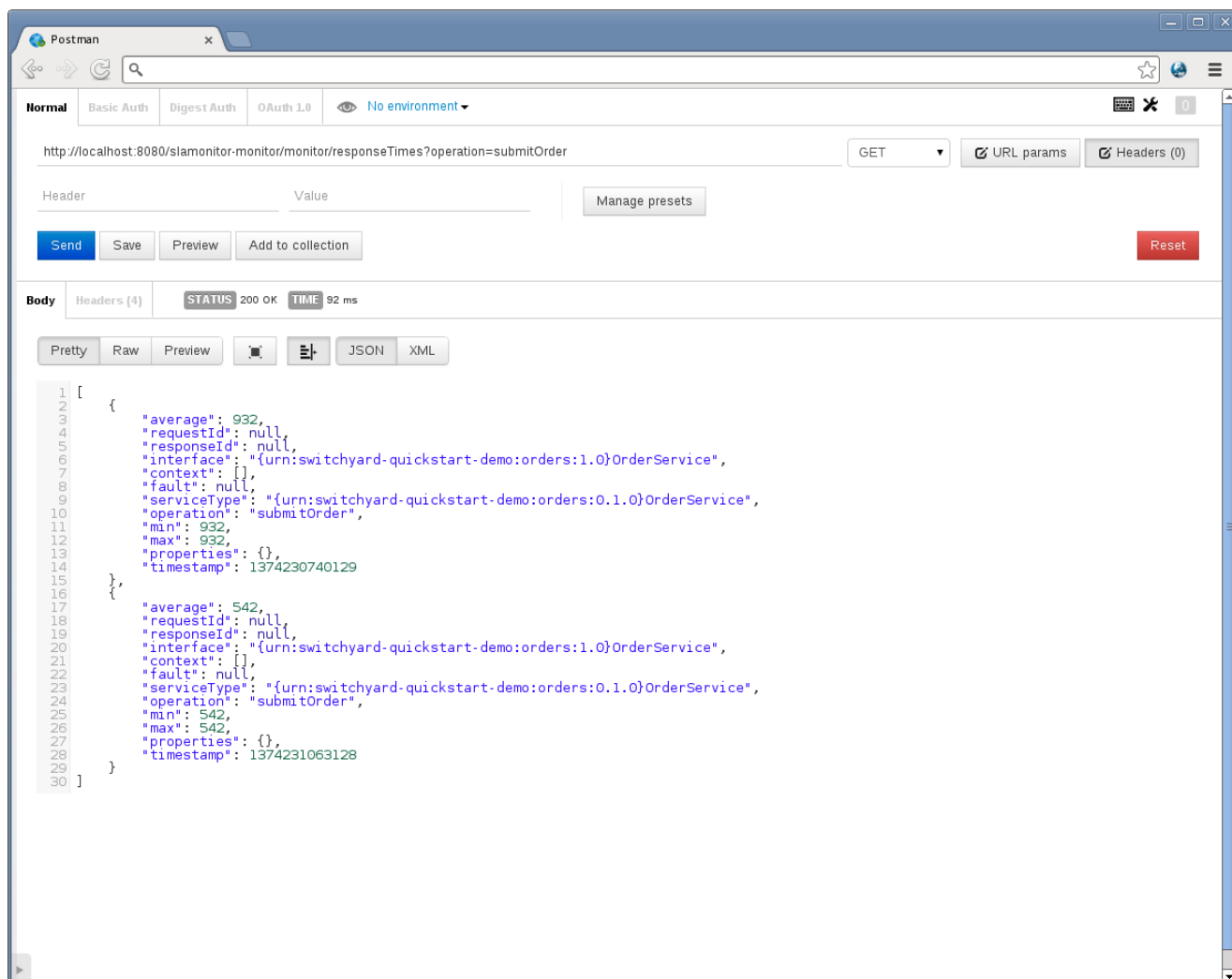
**Figure 3.4.**

> **Caution**
>
> If no query parameter is provided, then response times for all operations will be returned.

### 3.1.4. Summary

This quickstart demonstrates how Service Level Agreements can be policed using rules defined in an Event Processor Network, and reporting to end users using the pre-configured "Situations" active collection.

The rule used in this example is simple, detecting whether the response time associated with an operation on a service exceeds a particular level. However more complex temporal rules could be defined to identify the latency between any two points in a business transaction flow.