A Load Generation Framework for CEP Application Testing and Demos

(Note: This is a completely refactored and much enhanced version of the CEP Testing framework I developed some time ago. You can find the original description here: https://mojo.redhat.com/docs/DOC-971322)

1 Introduction

Have you ever lamented over how difficult it is to test a CEP application, let alone doing a demo? Common challenges include:

- How to generate events for testing a CEP application?
- How to demo a CEP application?
- Cannot be real-time, it takes too long
- Lack of infrastructure during demo
- Need repeatable outcome

In this article, I am going to show you a framework that I developed which allows you to define external events in a CSV file, play them back to your CEP application in demonstrations in accelerated time. This framework can also be used to generate a large volume of events based on event arrival distribution either in realtime or accelerated time to load test your CEP application. It is such a versatile tool that you can even use it to perform discrete event simulation (not described in this article).

The framework solves all the problems listed earlier by allowing you to:

- Configure load to drive your CEP application
- Run your CEP application in accelerated time
- See the results quickly
- Use it as a reusable infrastructure for CEP application testing and demos
- Achieve repeatable outcome

Examples will be provided to showcase the capabilities of the framework including playing back configured events for a CEP application and realtime load generation using JBoss Fuse/A-MQ and event arrival patterns (distributions).

This article is divided into the following main sections:

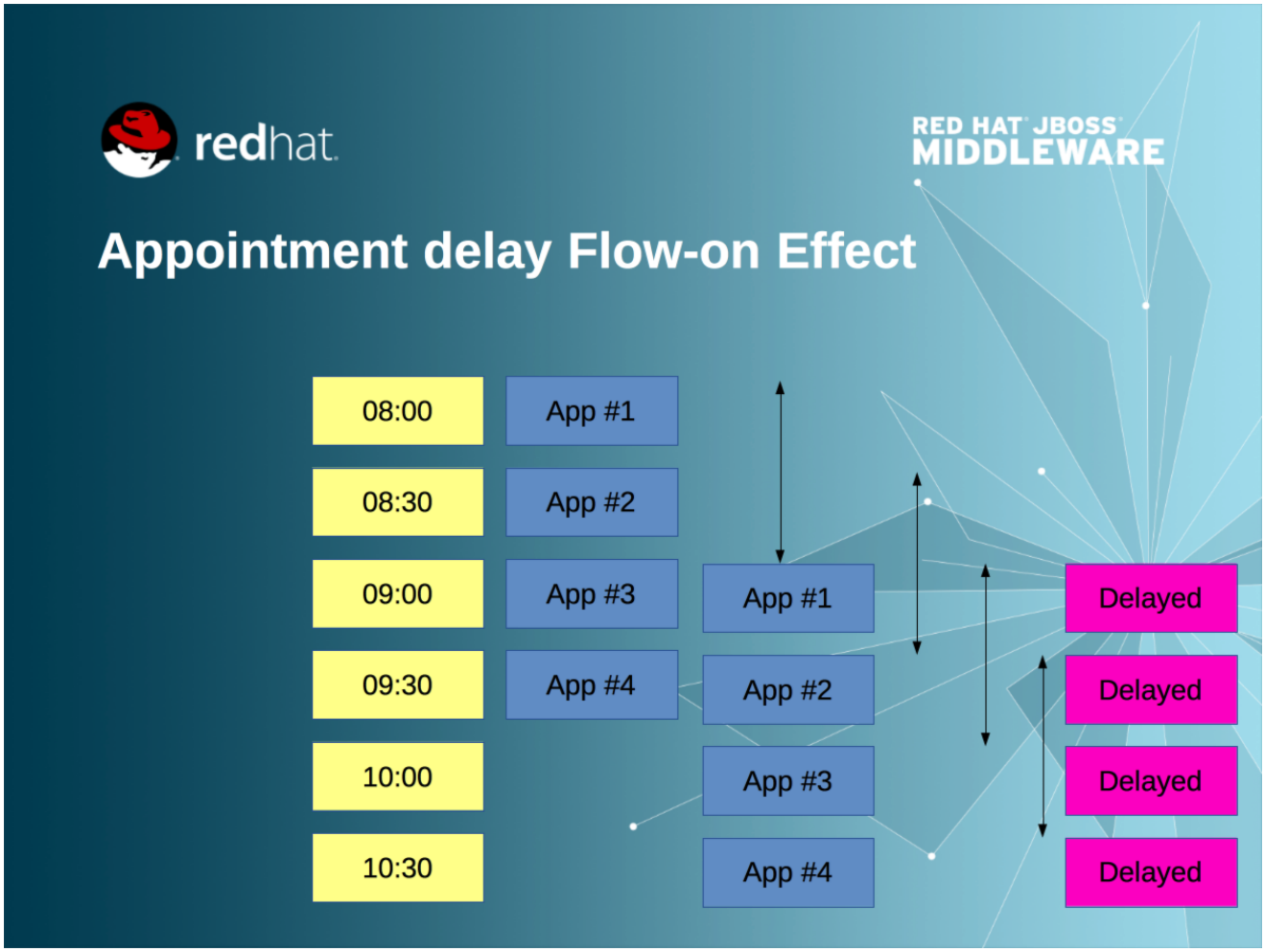
- The Optometrist CEP Application - this CEP application shows how to configure individual events to drive the CEP application in a CSV file.
- The Stock Price CEP Application - this is a simple CEP Application which illustrates the event generation capability based on event arrival distribution.

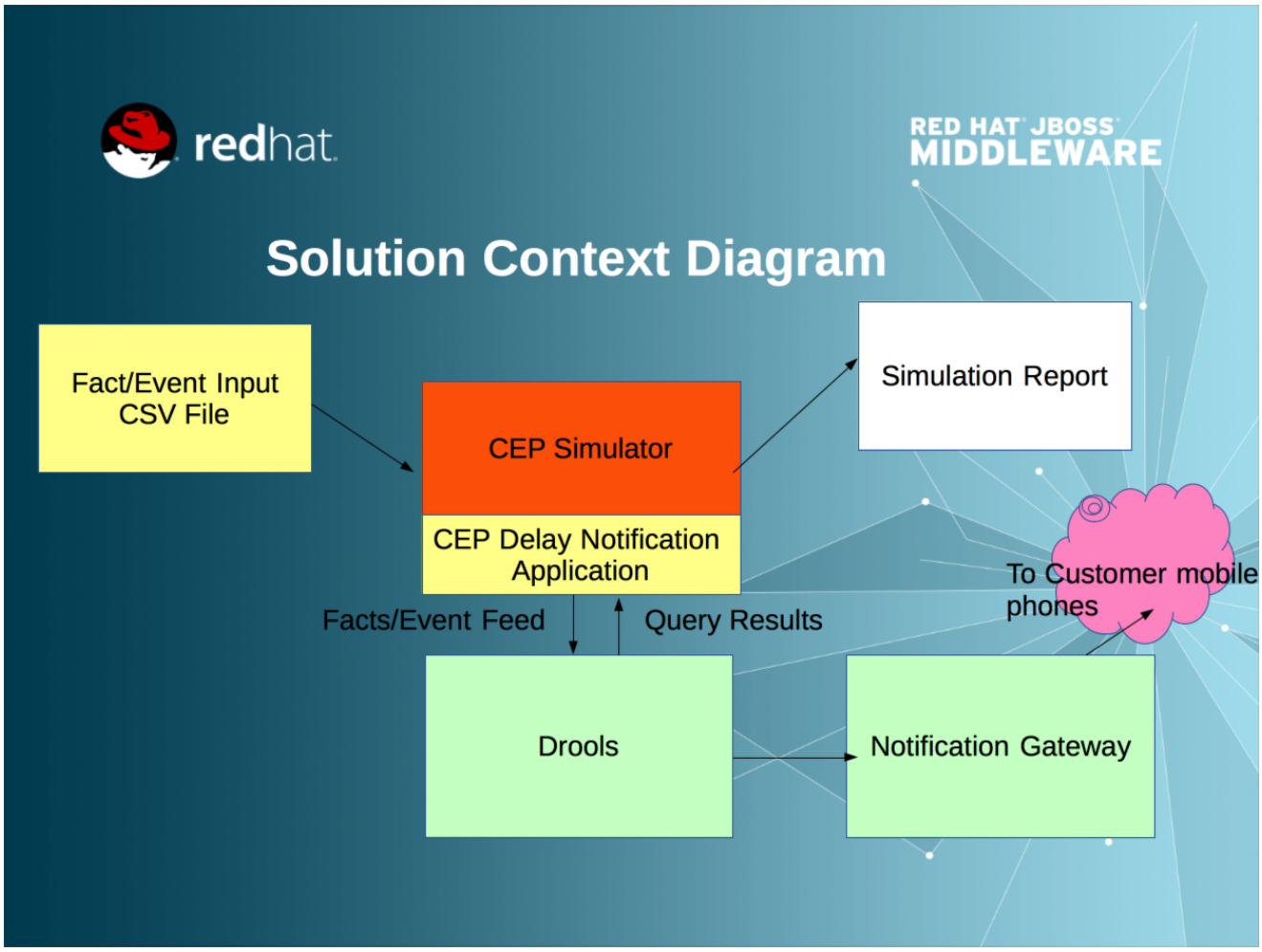
- Realtime Load Generation via Fuse and A-MQ integration - this section illustrates how realtime load generation can be achieved running multiple instances of the load generator to feed the Stock Price application via Fuse and A-MQ by applying software design patterns to loosely couple the load generator and the CEP application.
- How it works - shows the UML class diagram containing the load testing framework classes, their attributes, operations and relationships. It also describes how the framework works including how the load generator is loosely coupled to your CEP application using the Observer design pattern.

The first few sections give you an overview of the capabilities of the framework. The "How it works" section outlines how the framework works.

2 The Optometrist CEP Application

The Optometrist application is a simple CEP demo Application meant to enhance customer experience by notifying customers when their appointments have been delayed. The problem and solution are shown pictorially below.



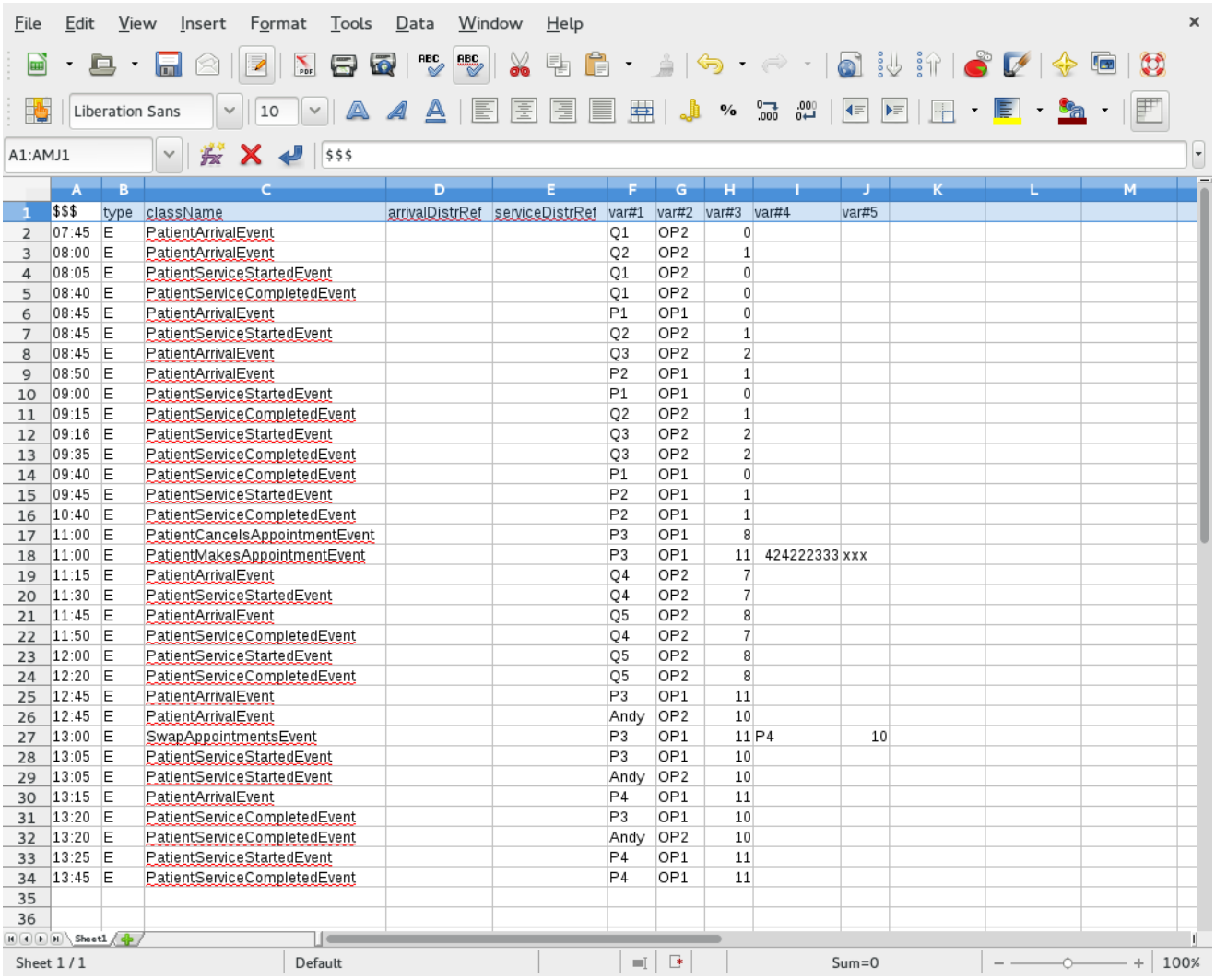


The framework provides a CepApplication base class which does most of the work for you. You just need to create your application as its subclass, create your event classes and define your events in a CSV file to be used by the load generator (DirectLoadGenerator). Your event classes need to be subclasses of the BaseEvent class to work with the load generator.

The partial source code of the Optometrist application and the accompanying CSV files that defines when events occur are shown below. Note that in this example, events are configured individually in the CSV file. Contrast this to that of using an arrival pattern in the next section.

```
// main program to setup and run the CEP application
  public static void main(String[] args) throws FileNotFoundException, IOException {

  // args[0] contains the csv file path info
  // creator is the object instantiator
  // KSESSION_NAME is the ksession name in the kmodule.xml in the META-INF directory
  // ENTRY_POINT is the event entry point name
  OptometristCepApplication simulator = new
OptometristCepApplication(KSESSION_NAME, ENTRY_POINT);
```

```
 simulator.setGlobal("notify", new PushOverNofification(APP_TOKEN));
 simulator.setGlobal("clock",simulator.getWallClock());
 insertFacts(simulator);

 ObservableEvent observableEvent = new ObservableEvent();
 observableEvent.addObserver(simulator);

 // set up fact/event object creator
 ObjectCreator creator = new ReflectionBasedObjectCreator(FACT_EVENT_PACKAGE);

 DirectLoadGenerator loadGenerator = new DirectLoadGenerator(new FileReader(args[0]),
creator, observableEvent);


 // run application
 loadGenerator.run();

 // output statistics for application
 simulator.printStats();

 // cleanup
 simulator.shutdown();
 }
```



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	$$$	type	className	arrivalDistrRef	serviceDistrRef	var#1	var#2	var#3	var#4	var#5			
2	07:45	E	PatientArrivalEvent			Q1	OP2	0					
3	08:00	E	PatientArrivalEvent			Q2	OP2	1					
4	08:05	E	PatientServiceStartedEvent			Q1	OP2	0					
5	08:40	E	PatientServiceCompletedEvent			Q1	OP2	0					
6	08:45	E	PatientArrivalEvent			P1	OP1	0					
7	08:45	E	PatientServiceStartedEvent			Q2	OP2	1					
8	08:45	E	PatientArrivalEvent			Q3	OP2	2					
9	08:50	E	PatientArrivalEvent			P2	OP1	1					
10	09:00	E	PatientServiceStartedEvent			P1	OP1	0					
11	09:15	E	PatientServiceCompletedEvent			Q2	OP2	1					
12	09:16	E	PatientServiceStartedEvent			Q3	OP2	2					
13	09:35	E	PatientServiceCompletedEvent			Q3	OP2	2					
14	09:40	E	PatientServiceCompletedEvent			P1	OP1	0					
15	09:45	E	PatientServiceStartedEvent			P2	OP1	1					
16	10:40	E	PatientServiceCompletedEvent			P2	OP1	1					
17	11:00	E	PatientCancelsAppointmentEvent			P3	OP1	8					
18	11:00	E	PatientMakesAppointmentEvent			P3	OP1	11	424222333	xxx			
19	11:15	E	PatientArrivalEvent			Q4	OP2	7					
20	11:30	E	PatientServiceStartedEvent			Q4	OP2	7					
21	11:45	E	PatientArrivalEvent			Q5	OP2	8					
22	11:50	E	PatientServiceCompletedEvent			Q4	OP2	7					
23	12:00	E	PatientServiceStartedEvent			Q5	OP2	8					
24	12:20	E	PatientServiceCompletedEvent			Q5	OP2	8					
25	12:45	E	PatientArrivalEvent			P3	OP1	11					
26	12:45	E	PatientArrivalEvent			Andy	OP2	10					
27	13:00	E	SwapAppointmentsEvent			P3	OP1	11	P4	10			
28	13:05	E	PatientServiceStartedEvent			P3	OP1	10					
29	13:05	E	PatientServiceStartedEvent			Andy	OP2	10					
30	13:15	E	PatientArrivalEvent			P4	OP1	11					
31	13:20	E	PatientServiceCompletedEvent			P3	OP1	10					
32	13:20	E	PatientServiceCompletedEvent			Andy	OP2	10					
33	13:25	E	PatientServiceStartedEvent			P4	OP1	11					
34	13:45	E	PatientServiceCompletedEvent			P4	OP1	11					
35													
36													

The output of the application is shown below:

```
                ...
[DEBUG] 13:25:00.000: Inserting event: PatientServiceCompletedEvent
[DEBUG] 13:45:00.000: Advancing clock by: 1200000 milliseconds
13:45:00.000: Patient P4 service completed, Scheduled start time: 13:30:00.000
Total Optometrist count: 2



---------------------------------------------
For Optometrist: OP1 of North Sydney Optometrists
Total Appointment count: 4
Customer =        P1, Wait Time =  60 minutes, ServiceTime =  40 minutes, Scheduled
StartTime = 08:00
```

```
Customer =        P2, Wait Time =  75 minutes, ServiceTime =  55 minutes, Scheduled
StartTime = 08:30
Customer =        P4, Wait Time =  -5 minutes, ServiceTime =  20 minutes, Scheduled
StartTime = 13:30
Customer =        P3, Wait Time =   5 minutes, ServiceTime =  15 minutes, Scheduled
StartTime = 13:00


Average Wait Time   =   33.8 minutes, Std. Dev. =   39.7 minutes
Average ServiceTime =   32.5 minutes, std. Dev. =   18.5 minutes
for a total of: 4 appointments


SMS events generated:
SMS: delay of     5 minutes sent at 13:05
SMS: delay of     0 minutes sent at 10:40
SMS: delay of    75 minutes sent at 09:45
SMS: delay of    60 minutes sent at 09:00


------------------------------------------------
For Optometrist: OP2 of Crows Nest Optometrists
Total Appointment count: 6
Customer =        Q1, Wait Time =   5 minutes, ServiceTime =  35 minutes, Scheduled
StartTime = 08:00
Customer =        Q2, Wait Time =  15 minutes, ServiceTime =  30 minutes, Scheduled
StartTime = 08:30
Customer =        Q3, Wait Time =  16 minutes, ServiceTime =  19 minutes, Scheduled
StartTime = 09:00
Customer =        Q4, Wait Time =   0 minutes, ServiceTime =  20 minutes, Scheduled
StartTime = 11:30
Customer =        Q5, Wait Time =   0 minutes, ServiceTime =  20 minutes, Scheduled
StartTime = 12:00
Customer =       Andy, Wait Time =   5 minutes, ServiceTime =  15 minutes, Scheduled
StartTime = 13:00


Average Wait Time   =    6.8 minutes, Std. Dev. =    7.1 minutes
Average ServiceTime =   23.2 minutes, std. Dev. =    7.6 minutes
for a total of: 6 appointments


SMS events generated:
```

```
SMS: delay of     0 minutes sent at 13:20
SMS: delay of     5 minutes sent at 13:05
SMS: delay of     0 minutes sent at 09:35
SMS: delay of    15 minutes sent at 09:15
SMS: delay of    15 minutes sent at 08:45
SMS: delay of     5 minutes sent at 08:05
```

The load generator instantiates your event objects based on the configuration in the CSV file by using the ReflectionBasedObjectCreator class that comes with the framework.

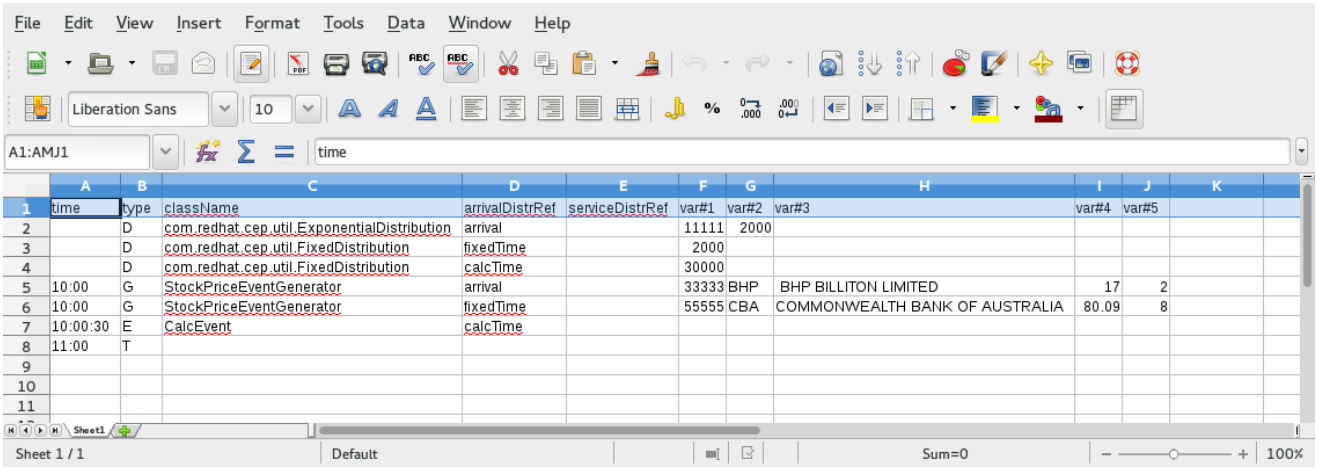
3 The Stock Price CEP Application

This is a very simple CEP application in which StockPriceEvents come in thick and fast. A CalcEvent arrives every 30 seconds. On receiving a CalcEvent, the CEP application calculates the min, max, average stock price and the total number of StockPriceEvents for each stock in the past 30 seconds (time window).
Unlike the previous example, the Stock Price CEP application uses the StockPriceEventGenerator to generate StockPriceEvents. The inter-arrival time of these events are based on the Normal Distribution (other distributions eg, uniform, exponential, fixed distributions are also supported).
The partial source code of the Stock Price CEP application , the accompanying CSV files that defines the event generation based on normal distribution and the StockPriceEventGenerator are shown below. Again, it uses the DirectLoadGenerator to generate load based on that defined in the CSV file.

```
                                        // main
                                        program
                                        to setup
                                        and run
                                        the CEP
                                        application

                                        public
                                        static void
                                        main(String[]
                                        args)
                                        throws
                                        FileNotFoundException,
                                        IOException
                                        {

                                                                // args[0]
                                                                contains
                                                                the csv
                                                                file path
                                                                info
```

```
StockPriceCepApplication
simulator
=
new
StockPriceCepApplication(KSESSION
ENTRY_POINT);



simulator.setGlobal("tickers",
new
HashSet(Arrays.asList(new
String[]
{ "BHP",
"CBA" })));

List<String>
list
=
new
ArrayList<String>();

simulator.setGlobal("list",
list);

simulator.setGlobal("clock",
simulator.getWallClock());

//
change
CSV
file
delimiter
(default
is
comma)

//
simulator.changeDelimiter('\t');

ObservableEvent
observableEvent
=
new
ObservableEvent();

observableEvent.addObserver(simula

//
set
up
fact/
event
object
creator
```

```
                                                                ObjectCreator
                                                                creator
                                                                =
                                                                new
                                                                ReflectionBasedObjectCreator(FACT_

                                                                DirectLoadGenerator
                                                                loadGenerator
                                                                =
                                                                new
                                                                DirectLoadGenerator(new
                                                                FileReader(args[0]),
                                                                creator,
                                                                observableEvent);



                                                                //
                                                                run
                                                                application
                                                                loadGenerator.run();

                                                                //
                                                                output
                                                                statistics
                                                                for
                                                                application
                                                                simulator.printStats(list);

                                                                //
                                                                cleanup
                                                                simulator.shutdown();

                                                    }
```



	A	B	C	D	E	F	G	H	I	J	K
1	time	type	className	arrivalDistrRef	serviceDistrRef	var#1	var#2	var#3	var#4	var#5	
2		D	com.redhat.cep.util.ExponentialDistribution	arrival		11111	2000				
3		D	com.redhat.cep.util.FixedDistribution	fixedTime		2000					
4		D	com.redhat.cep.util.FixedDistribution	calcTime		30000					
5	10:00	G	StockPriceEventGenerator	arrival		33333	BHP	BHP BILLITON LIMITED	17	2	
6	10:00	G	StockPriceEventGenerator	fixedTime		55555	CBA	COMMONWEALTH BANK OF AUSTRALIA	80.09	8	
7	10:00:30	E	CalcEvent	calcTime							
8	11:00	T									
9											
10											
11											


```
public class StockPriceEventGenerator implements EventGenerator {
```

```
 private String symbol;
 private String companyName;
 private double price;
 private NormalDistribution distr;

 public StockPriceEventGenerator(int seed, String symbol, String companyName, double
price, double priceDifference) {
 this.symbol = symbol;
 this.companyName = companyName;
 this.price = price;


 distr = new NormalDistribution(seed, priceDifference, priceDifference);
 }

 public BaseEvent generateEvent() {


 return new StockPriceEvent(
 symbol,
 companyName,
 price + distr.sample()
 );
 }
```

The output of the application is shown below:

```
...
[DEBUG] 10:59:53.333: Advancing clock by: 1333 milliseconds
[DEBUG] Scheduling event - com.redhat.cep.stockprice.model.StockPriceEvent:
11:00:00.657 : arrival :  : true - symbol=BHP, companyName=BHP BILLITON LIMITED,
price=   17.38, 7324 millisec
[DEBUG] scheduled at - 10:59:53.333
[DEBUG] 10:59:54.000: Advancing clock by: 667 milliseconds
[DEBUG] Scheduling event - com.redhat.cep.stockprice.model.StockPriceEvent:
10:59:56.000 : fixedTime :  : true - symbol=CBA, companyName=COMMONWEALTH
BANK OF AUSTRALIA, price=   99.47, 2000 millisec
[DEBUG] scheduled at - 10:59:54.000
[DEBUG] 10:59:56.000: Advancing clock by: 2000 milliseconds
```

```
[DEBUG] Scheduling event - com.redhat.cep.stockprice.model.StockPriceEvent:
10:59:58.000 : fixedTime :  : true - symbol=CBA, companyName=COMMONWEALTH
BANK OF AUSTRALIA, price=   79.28, 2000 millisec
[DEBUG] scheduled at - 10:59:56.000
[DEBUG] 10:59:58.000: Advancing clock by: 2000 milliseconds
[DEBUG] Scheduling event - com.redhat.cep.stockprice.model.StockPriceEvent:
11:00:00.000 : fixedTime :  : true - symbol=CBA, companyName=COMMONWEALTH
BANK OF AUSTRALIA, price=   88.52, 2000 millisec
[DEBUG] scheduled at - 10:59:58.000
[DEBUG] 11:00:00.000: Advancing clock by: 2000 milliseconds
[DEBUG] Scheduling event - com.redhat.cep.stockprice.model.CalcEvent: 11:00:30.000 :
calcTime :  : true, 30000 millisec
[DEBUG] scheduled at - 11:00:00.000
11:00:00.000 Stock CBA: avg price=  90.368, min price=  77.849, max price= 101.091
samples=    14
11:00:00.000 Stock BHP: avg price=  19.179, min price=  16.728, max price=  21.542
samples=    12
List size: 240
10:00:30.000 Stock CBA: avg price=  84.687, min price=  67.690, max price=  99.599
samples=    14
10:00:30.000 Stock BHP: avg price=  18.579, min price=  15.024, max price=  23.125
samples=    17
10:01:00.000 Stock CBA: avg price=  88.827, min price=  77.905, max price= 100.252
samples=    14
10:01:00.000 Stock BHP: avg price=  19.083, min price=  14.748, max price=  22.546
samples=    21
10:01:30.000 Stock CBA: avg price=  87.800, min price=  72.108, max price= 103.527
samples=    14
10:01:30.000 Stock BHP: avg price=  19.811, min price=  17.946, max price=  24.794
samples=    15
10:02:00.000 Stock CBA: avg price=  86.496, min price=  77.187, max price=  98.792
samples=    15
10:02:00.000 Stock BHP: avg price=  18.833, min price=  16.216, max price=  22.199
samples=    15
10:02:30.000 Stock CBA: avg price=  88.079, min price=  67.132, max price= 102.592
samples=    15
10:02:30.000 Stock BHP: avg price=  19.368, min price=  15.774, max price=  23.830
samples=    18
10:03:00.000 Stock CBA: avg price=  86.700, min price=  75.992, max price=  94.309
samples=    15
10:03:00.000 Stock BHP: avg price=  19.479, min price=  14.724, max price=  22.770
samples=    14
```

```
10:03:30.000 Stock CBA: avg price=  85.911, min price=  74.589, max price= 105.394
samples=    14
10:03:30.000 Stock BHP: avg price=  18.754, min price=  15.875, max price=  22.568
samples=    17
10:04:00.000 Stock CBA: avg price=  86.873, min price=  76.159, max price=  98.093
samples=    14
10:04:00.000 Stock BHP: avg price=  18.546, min price=  15.043, max price=  21.758
samples=    17
10:04:30.000 Stock CBA: avg price=  89.379, min price=  73.065, max price= 109.865
samples=    15
10:04:30.000 Stock BHP: avg price=  18.677, min price=  15.091, max price=  24.128
samples=    16
...
```

4 Realtime Load Generation via Fuse and A-MQ Integration

This example reuses the Stock Price CEP application. Instead of using the DirectLoadGenerator, it uses the JmsLoadGenerator. Multiple instances of the JmsLoadGenerator can be set up to generate events and send them to a message queue if, for example, one instance is not sufficient to generate the required load. The Stock Price CEP application is configured as part of a Camel route that picks up events from the message queue and processes them.

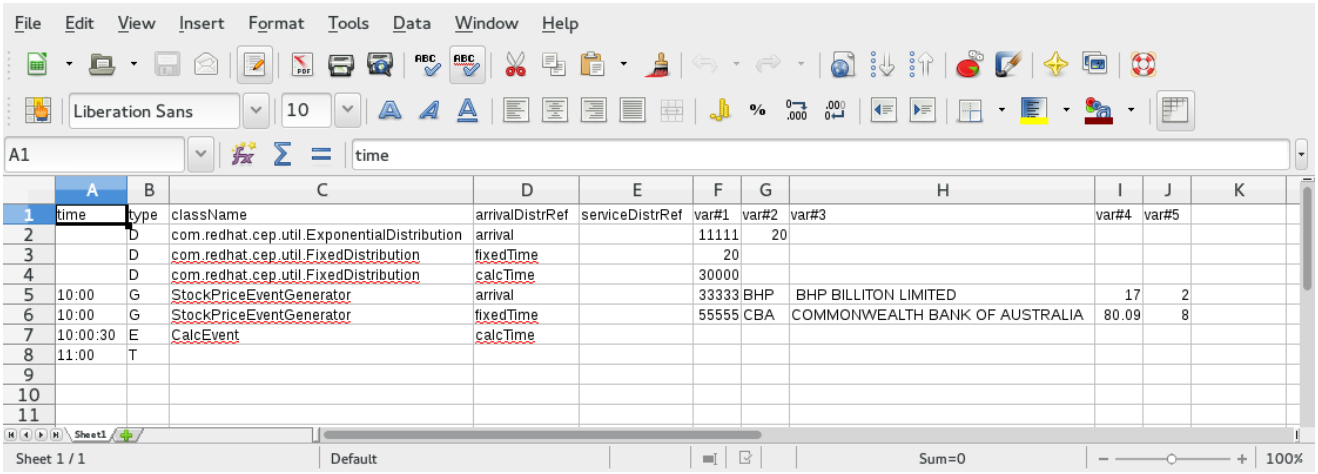
The partial source code of the Stock Price CEP Camel route and the accompanying CSV files that define the event generation based on normal distribution are shown below. This time it uses the JmsLoadGenerator instead of the DirectLoadGenerator to generate the load. The JmsLoadGenerator is a simple subclass of AbstractLoadGenerator. It uses Camel's ProducerTemplate to send messages to a message queue. Two instances of the JmsLoadGenerator were used to illustrate use of multiple load generator instances for realtime load generation.

```
public class JmsLoadGenerator extends AbstractLoadGenerator {
  static Logger logger = Logger.getLogger(JmsLoadGenerator.class);



  static final int PROGRESS_COUNT = 1000;

  private ProducerTemplate camelTemplate;
  private int count = 0;



  public JmsLoadGenerator(FileReader reader, ObjectCreator creator, ProducerTemplate
camelTemplate)
```

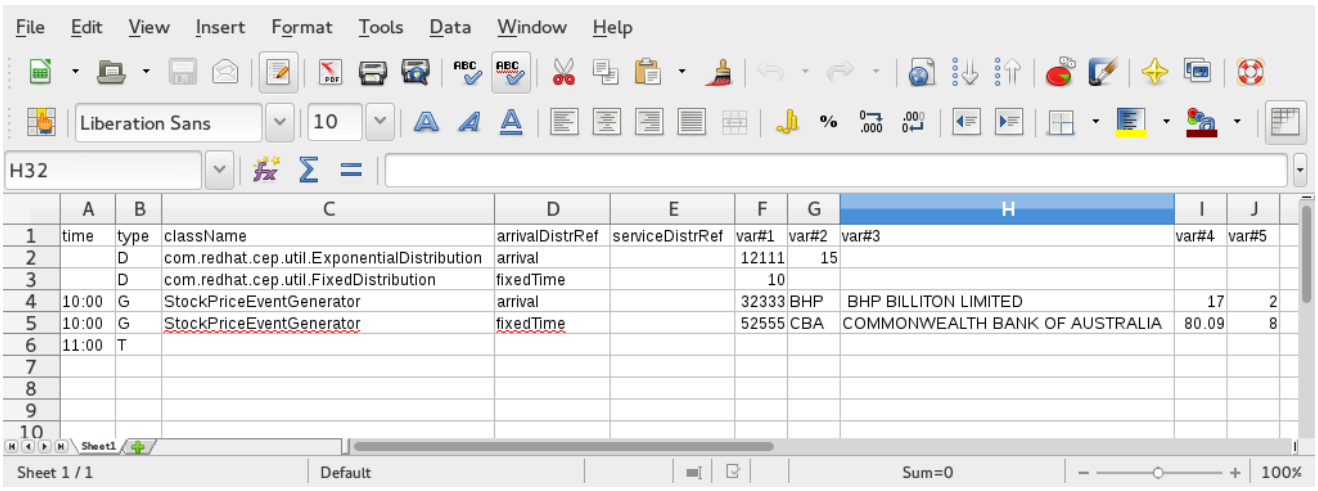
```
 throws IOException {
 super(reader, creator);
 this.camelTemplate = camelTemplate;
 }




 @Override
 public void processEvent(BaseEvent event, long millisSinceLastEvent) {
 if (millisSinceLastEvent > 0) {
 try {
 Thread.sleep(millisSinceLastEvent);
 } catch (InterruptedException e) {
 // TODO Auto-generated catch block
 e.printStackTrace();
 }

 }




 camelTemplate.sendBody("jms:queue:events", ExchangePattern.InOnly, event);
 if ((++count % PROGRESS_COUNT) == 0) {
 logger.info(count + " events sent...");
 }




 }

 //
 /**
 *
 * This is a Camel client using ProducerTemplate to send events to a message queue
 * args[0] contains the event package name eg, com.redhat.cep.stockprice.model where
most events and
 *         other objects are defined
 * args[1] contains the csv file path info
 */
   public static void main(final String[] args) throws Exception {
```

```
        logger.warn("JmsLoadGenerator requires that the CamelServer is running already!");


        AbstractApplicationContext context = new
ClassPathXmlApplicationContext("JmsLoadGenerator.xml");


        // use camel ProducerTemplate
        ProducerTemplate camelTemplate = context.getBean("camelTemplate",
ProducerTemplate.class);


  // set up fact/event object creator
  ObjectCreator creator = new ReflectionBasedObjectCreator(args[0]);

  // args[0] contains the csv file path info
  JmsLoadGenerator loadGen = new JmsLoadGenerator(new FileReader(args[1]), creator,
camelTemplate);

  logger.info("start sending events...");

  loadGen.run();




        // close the application context
        context.close();

        logger.info("Finished execution.");
    }
```

Each instance of the JmsLoadGenerator uses its own CSV configuration file:



	A	B	C	D	E	F	G	H	I	J	K
1	time	type	className	arrivalDistrRef	serviceDistrRef	var#1	var#2	var#3	var#4	var#5	
2		D	com.redhat.cep.util.ExponentialDistribution	arrival		11111	20				
3		D	com.redhat.cep.util.FixedDistribution	fixedTime		20					
4		D	com.redhat.cep.util.FixedDistribution	calcTime		30000					
5	10:00	G	StockPriceEventGenerator	arrival		33333	BHP	BHP BILLITON LIMITED	17	2	
6	10:00	G	StockPriceEventGenerator	fixedTime		55555	CBA	COMMONWEALTH BANK OF AUSTRALIA	80.09	8	
7	10:00:30	E	CalcEvent	calcTime							
8	11:00	T									
9											
10											
11											



	A	B	C	D	E	F	G	H	I	J
1	time	type	className	arrivalDistrRef	serviceDistrRef	var#1	var#2	var#3	var#4	var#5
2		D	com.redhat.cep.util.ExponentialDistribution	arrival		12111	15			
3		D	com.redhat.cep.util.FixedDistribution	fixedTime		10				
4	10:00	G	StockPriceEventGenerator	arrival		32333	BHP	BHP BILLITON LIMITED	17	2
5	10:00	G	StockPriceEventGenerator	fixedTime		52555	CBA	COMMONWEALTH BANK OF AUSTRALIA	80.09	8
6	11:00	T								
7										
8										
9										
10										

The Camel route and the StockPriceApplicationAdapter are shown below:

```
public class ServerRoutes extends RouteBuilder {


   @Override
   public void configure() throws Exception {
         // route from the events queue to our CEP application which is a spring bean
registered with id=processEvent


   from("jms:queue:events").to("processEvent");


   }
```

```
@Service(value = "processEvent")
public class StockPriceApplicationAdapter {
```

```
 private ObservableEvent observableEvent;
 private StockPriceCepApplication simulator;

 public StockPriceApplicationAdapter() {


 // KSESSION_NAME is the ksession name in the kmodule.xml in the META-INF directory
 // ENTRY_POINT is the event entry point name


 simulator = new StockPriceCepApplication(StockPriceCepApplication.KSESSION_NAME,
StockPriceCepApplication.ENTRY_POINT);
 simulator.setSystemTimeMode();


 simulator.setGlobal("tickers", new HashSet(Arrays.asList(new String[] { "BHP", "CBA" })));
 List<String> list = new ArrayList<String>();
 simulator.setGlobal("list", list);
 simulator.setGlobal("clock", simulator.getWallClock());

 // change CSV file delimiter (default is comma)
 //simulator.changeDelimiter('\t');

 observableEvent = new ObservableEvent();
 observableEvent.addObserver(simulator);


 }

 public void processEvent(BaseEvent event) {
 observableEvent.setValue(event);


 }


}
```

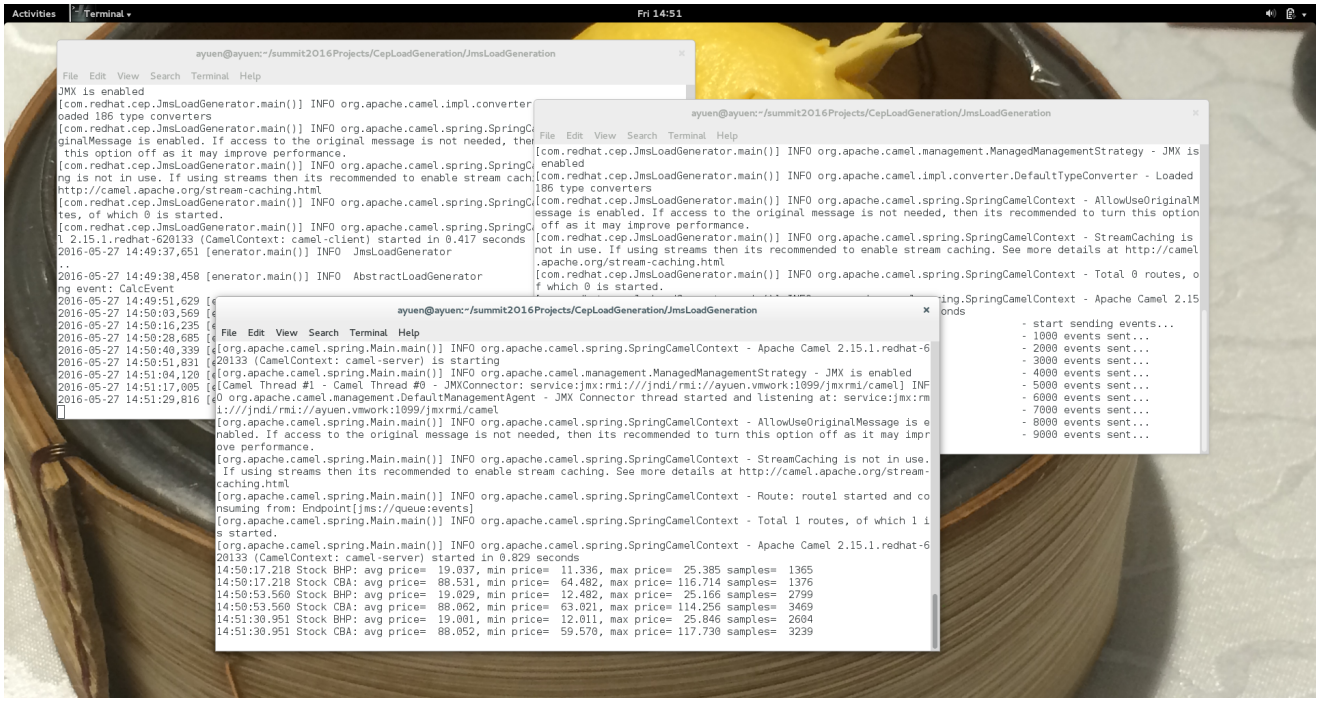
The realtime load testing is run in 3 separate command prompts. Run one command in each command prompt:

mvn exec:java -PCamelServer

mvn exec:java -PJmsLoadGenerator

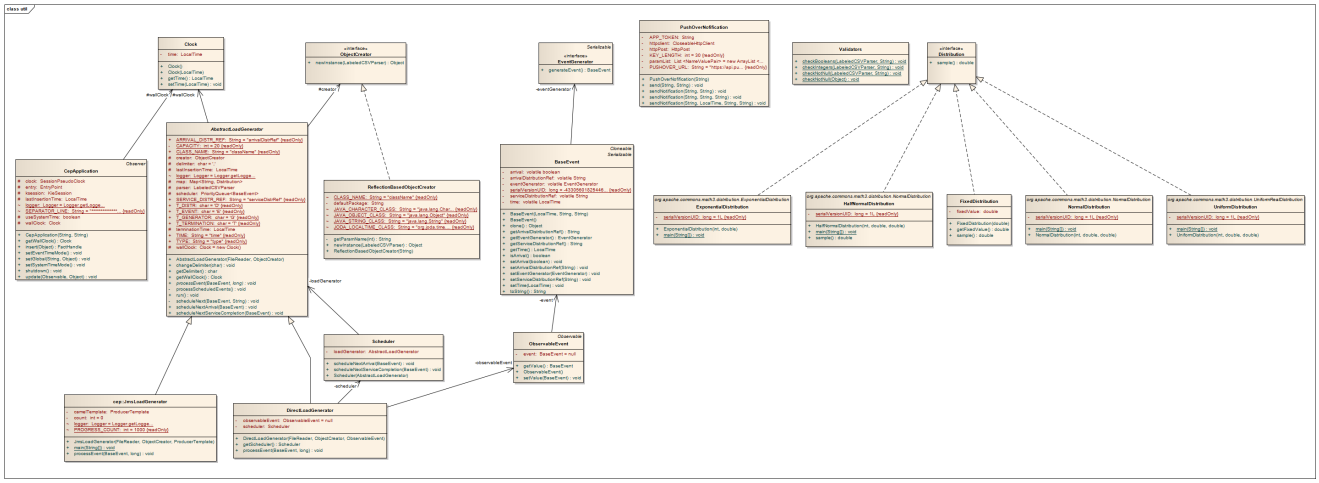
mvn exec:java -PJmsLoadGenerator2

The screen captured while running the load testing is shown below:



5 How it Works

The UML class diagram showing the load testing framework classes, their attributes, operations and relationships is shown below.



The two main components of the framework are: load generator and your CEP application. They are loosely coupled together using the Observer design pattern (Java's Observer and Observable interfaces). Your CEP application creates an ObservableEvent and adds itself as an observer. Note that your CEP application's base class implements the Observer interface. It thens instantiates a load generator by passing the ObservableEvent object as one of its parameters. Every time the load generator wants to send an event to the CEP application, it invokes its implementation of its base class' (AbstractLoadGenerator) processEvent interface. The implementation of the processEvent interface is different for the different load generators: DirectLoadGenerator and JmsLoadGenerator.
For DirectLoadGenerator, processEvent simply calls ObservableEvent's setValue method which, in turn, calls setChanged() and notifyObservers(). Since your CEP application has registered itself as an observer when creating the ObservableEvent object, it will get notified whenever an event is available.

```
// main program to setup and run the CEP application
  public static void main(String[] args) throws FileNotFoundException, IOException {



  // args[0] contains the csv file path info
  // creator is the object instantiator
  // KSESSION_NAME is the ksession name in the kmodule.xml in the META-INF directory
  // ENTRY_POINT is the event entry point name
  OptometristCepApplication simulator = new
OptometristCepApplication(KSESSION_NAME, ENTRY_POINT);
  simulator.setGlobal("notify", new PushOverNofification(APP_TOKEN));
  simulator.setGlobal("clock",simulator.getWallClock());
  insertFacts(simulator);



  ObservableEvent observableEvent = new ObservableEvent();
  observableEvent.addObserver(simulator);



  // set up fact/event object creator
  ObjectCreator creator = new ReflectionBasedObjectCreator(FACT_EVENT_PACKAGE);



  DirectLoadGenerator loadGenerator = new DirectLoadGenerator(new FileReader(args[0]),
creator, observableEvent);
```

```
// run application
loadGenerator.run();



// output statistics for application
simulator.printStats();



// cleanup
simulator.shutdown();
}
```

For JmsLoadGenerator, processEvent uses Camel's ProducerTemplate to send the events to a message queue. Your CEP application is set up as part of a Camel route which picks up messages from the message queue and passes them on to your CEP application for processing via the StockPriceApplicationAdapter's processEvent service which, again, uses observerableEvent's setValue method to notify your CEP application. The JmsLoadGenerator and your CEP application are loosely coupled together via a message queue and the Observer design pattern.

```
                                                    //
                                                    /**
                                                    *
                                                    * This is a Camel client
                                                    using ProducerTemplate
                                                    to send events to a
                                                    message queue
                                                    * args[0] contains the
                                                    event package name eg,
                                                    com.redhat.cep.stockprice.model
                                                    where most events and
                                                    *       other objects are
                                                    defined
                                                    * args[1] contains the csv
                                                    file path info
                                                    */
  public static void main(final String[] args) throws Exception {



     logger.warn("JmsLoadGenerator requires that the CamelServer is running already!");
```

```
        AbstractApplicationContext context = new
ClassPathXmlApplicationContext("JmsLoadGenerator.xml");



        // use camel ProducerTemplate
        ProducerTemplate camelTemplate = context.getBean("camelTemplate",
ProducerTemplate.class);

                                                                    // set up
                                                                    fact/event
                                                                    object
                                                                    creator

                                                                    ObjectCreator
                                                                    creator
                                                                    = new
                                                                    ReflectionBasedObjectCreator(args[0]);

                                                                    // args[0]
                                                                    contains
                                                                    the csv
                                                                    file path
                                                                    info

                                                                    JmsLoadGenerator
                                                                    loadGen
                                                                    = new
                                                                    JmsLoadGenerator(new
                                                                    FileReader(args[1]),
                                                                    creator,
                                                                    camelTemplate);

                                                                    logger.info("start
                                                                    sending
                                                                    events...");

                                                                    loadGen.run();




        // close the application context
        context.close();

        logger.info("Finished execution.");
    }
```

6 Conclusion

The various examples provided showed how easy it is to generate load for testing your CEP application using my load testing framework. You can either configure individual events to occur at specific times or configure the framework to generate events based on a particular arrival pattern or distribution. In addition, you can use either the DirectLoadGenerator to generate the events as specified in a CSV file in accelerated time or use one or more instances of the JmsLoadGenerator to generate realtime test load based on an arrival pattern. Your CEP application can switch between using either the DirectLoadGenerator or the JmsLoadGenerator with minor changes . The functionality and other convenience classes (including distribution functions, simple notification services to iPhone or Android phones, etc.) will allow you to demonstrate your CEP application to your customers in accelerated time. It will also help you load test your CEP application as part of your development process.

I am in the process of cleaning up the code and making it available on GitHub so stay tuned. I shall update this document with the link to the source code soon.