Red Hat Reference Architecture Series

# Highly-Available Complex Event Processing with Red Hat JBoss BRMS

Jeremy Ary, Ugo Landini, Fabio Marinelli

Version 1.1, August 2016

# Table of Contents


Comments and Feedback . . . 2
1. Executive Summary . . . 3
2. Highly-Available & Scalable Complex Event Processing with Red Hat JBoss BRMS . . . 4
  2.1. Overview . . . 4
    2.1.1. Features . . . 4
    2.1.2. Statistical Use Case . . . 5
  2.2. Architectural Design . . . 6
    2.2.1. Event Channel . . . 6
    2.2.2. HACEP Node Structure . . . 7
    2.2.3. Event Consumption . . . 8
    2.2.4. Session Instantiation, Resumption, and State Replication . . . 8
    2.2.5. Fault Tolerance . . . 9
    2.2.6. Session Snapshotting . . . 9
3. Reference Architecture Environment . . . 11
  3.1. Overview . . . 11
  3.2. JBoss EAP 7.0.1 . . . 11
  3.3. HACEP Cluster Nodes . . . 11
  3.4. HACEP Core Components . . . 12
    3.4.1. JGroups . . . 12
    3.4.2. Routing . . . 12
    3.4.3. JBoss BRMS . . . 12
    3.4.4. Data Grid . . . 12
  3.5. Event Channel Components . . . 13
    3.5.1. JBoss A-MQ 6.2.1 Cluster . . . 13
    3.5.2. ZooKeeper Cluster . . . 13
4. Creating the Environment . . . 14
  4.1. Prerequisites . . . 14
  4.2. Downloads . . . 14
  4.3. Installation . . . 14
    4.3.1. ZooKeeper Ensemble . . . 14
    4.3.2. JBoss A-MQ Cluster . . . 16
    4.3.3. EAP 7.0.1 . . . 17
  4.4. Conclusion . . . 23
5. Design and Development . . . 24
  5.1. HACEP Integration and Source Examples . . . 24
  5.2. Project Setup . . . 24

5.3. Running the CLI HACEP Example . . . 25
5.4. Running the EAP HACEP Example . . . 27
5.5. Integrating HACEP into an Application . . . 28
5.5.1. Parent Project Configuration . . . 28
5.5.2. The integration-model Module . . . 36
5.5.3. The integration-rules Module . . . 42
5.5.4. The purchase-publisher Module . . . 46
5.5.5. The integration-app Module . . . 50
5.6. Packaging & Deploying the Integration Application . . . 60
5.7. Executing the Purchase Publisher for Real-Time Observation . . . 60
5.8. Summation . . . 60
6. Conclusion . . . 61
Appendix A: Revision History . . . 62
Appendix B: Contributors . . . 63

100 East Davie Street
Raleigh NC 27601 USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588
Research Triangle Park NC 27709 USA

Linux is a registered trademark of Linus Torvalds. Red Hat, Red Hat Enterprise Linux and the Red Hat "Shadowman" logo are registered trademarks of Red Hat, Inc. in the United States and other countries. Microsoft and Windows are U.S. registered trademarks of Microsoft Corporation. UNIX is a registered trademark of The Open Group. Intel, the Intel logo and Xeon are registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. OpenStack is the trademark of the OpenStack Foundation. All other trademarks referenced herein are the property of their respective owners.

© 2016 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the security@redhat.com key is: CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

# Comments and Feedback

In the spirit of open source, we invite anyone to provide feedback and comments on any of the reference architectures. Although we review our papers internally, sometimes issues or typographical errors are encountered. Feedback allows us to not only improve the quality of the papers we produce, but allows the reader to provide their thoughts on potential improvements and topic expansion to the papers. Feedback on the papers can be provided by emailing refarch-feedback@redhat.com. Please refer to the title within the email.

# 1. Executive Summary

Oftentimes, when working with records of significant change of state in the application domain at a given point in time, known as *events*, it becomes necessary to identify and analyze patterns in such a way as to deduce higher-level impacts, time-based correlations, and relationships that can factor into or directly affect business processes, application code, decision-making, and the like. As a quick example, consider the gaming industry, where the actions performed by a player or set of players must be processed quickly and synchronously in order to best process their intent. The financial sector also lends itself to a simplified use case, wherein credit card transactions over a given period of time can be analyzed in order to assist in fraud detection and prevention. This process of detecting and selecting interesting events from within an event cloud, finding their relationships, and inferring new data is known as **Complex Event Processing**, or **CEP**.

When dealing with large amounts of events in a synchronous way, or even when considering time correlations amongst them, it's important to recognize high-availability as a requirement for an enterprise application related to CEP. Without consistency and redundancy, significant events can be overlooked or missed entirely, thus creating the possibility of exclusions or faulty assumptions concluded from processing an *event cloud*, or group of events.

Likewise, when considering the sheer volume of events possible within various application spaces, which still must be consumed in a consistent and reliable fashion, scalability becomes a factor. While early on, it's possible to simply increase hardware capacity for short-term gains, eventually, this model becomes unfeasible and thus, partitioning of capabilities and processing becomes a necessity.

This reference architecture reviews the `HACEP`, or Highly-Available and Scalable Complex Event Processing, framework and walks through the deployment and implementation of a few simplistic examples showcasing the various features that the framework offers. The goal of this reference architecture is to provide a thorough description of the steps required for establishing a highly-available, scalable, and CEP-capable architecture stack, while citing the rationale for inclusion of various toolings or patterns when applicable and the challenges overcome by utilizing the provided HACEP framework.

# 2. Highly-Available & Scalable Complex Event Processing with Red Hat JBoss BRMS

## 2.1. Overview

`HACEP` (pronounced *hä-sep*) is a scalable and highly-available architecture framework for enterprise-level usage of the complex event processing abilities in `Red Hat JBoss BRMS` (Business Rules Management System). HACEP combines `Red Hat JBoss Data Grid (JDG)`, `Apache Camel` capabilities as delivered in `JBoss Fuse`, `Red Hat JBoss BRMS`, and consumption of events from a Java Messaging Service (JMS) such as `Red Hat JBoss A-MQ` in a unique way to obtain horizontal scalability and failover.

HACEP is a generic solution for imposing a partitioning criteria, while no further constraints or limitations on capabilities are introduced. The framework is designed on a few fundamental patterns:

- *sharding*: horizontal partitioning, or splitting of data and workload into separated nodes
- *data affinity*: colocation of data and calculations related to said data together on a single owner node
- *event sourcing*: capturing every change of state within the application as an event object and storing them in the same sequence as they are received and applied for the lifespan duration of the application itself, thus allowing such events to be queried or utilized to rebuild current or past states

### 2.1.1. Features

Out of the box, BRMS CEP functionality doesn't account for high-availability or horizontal scaling. Introducing the HACEP framework to an enterprise architectural stack fulfills these functional needs given that it can easily scale from 2 to 100s of nodes and back again, even dynamically at runtime. The clusters themselves are inherently highly-available when provided with the minimal 2 nodes per cluster. Outside of scaling and fault tolerance, some of the other features gained from usage of the framework include:

- survival in cases of multiple node failures
- in-memory read/write performance for extreme throughput
- dynamic CEP rules recompilation
- multiple disk storage options
- minimal system footprint
- rolling upgrade support
- plain JVM and `Red Hat JBoss Enterprise Application Platform (EAP)` support

### 2.1.2. Statistical Use Case

As mentioned before, CEP can be summarized as the process of detecting and selecting interesting events from within an event cloud, finding their relationships, and inferring new data based on the findings. In more practical terms, consider a system with users who interact with an application several times a day for a given consecutive number of days. As an example, listed below are a few possible statistical statements of interest involving a hypothetical gaming application:

- User performs an action *T* times
- User performs an action *T* times for *D* consecutive days
- User places *X* actions/items in *D* days
- User wins/loses more than *N* times
- User wins/loses a cumulative *S* amount

Each and every event fed to the system is possibly a point of interest, so all events must be captured, considered, and stored away for fault tolerance. Network traffic considerations aside, a singular system utilizing `JBoss BRMS` sessions to handle such a workload would quickly necessitate scaling of session memory to an unreasonable point. When thinking in terms of numbers, such a system could potentially have a very large workload to deal with in an isolated fashion. In order to better visualize the need to compensate for the significant amount of events and facts involved, consider a probable workload such as the one given below:

- 10 million events per day
- 1 million registered users
- approximate 30-day sliding time window
- 8 thousand concurrent users per day
- 90 thousand unique users over a 30-day period
- approximately 200 bytes per event

Given that JBoss BRMS CEP functionality doesn't provide for scalability out of the box, at approximately 200 bytes per event, 10 million events per day over a period of 30 days, just the size of raw event facts alone reaches 60GB, which doesn't account for a significant amount of information other than event facts that a BRMS session would be responsible for. At a minimum, such a system would call for 128GB of heap space in order to store everything required within a single rules session.

Introducing HACEP's capabilities for scaling and partitioning via data affinity dramatically decreases the memory requirements while also introducing asynchronous session processing. A 4-node HACEP cluster with 8GB heap per node can easily deal with the 10-thousand concurrent users and time window of 30 days. If around 10% of concurrent users were generating an event every 3 seconds, the rules sessions within the HACEP nodes would be approximately 5MB each, which, alongside the heap allotment, is a drastic improvement on memory allocation alone. Paired with gains in performance and fault tolerance, the benefits of HACEP become self-evident.

## 2.2. Architectural Design

At a high level, an enterprise system designed for usage of HACEP generally consists of two major components, the **Event Channel**, responsible for feeding the HACEP cluster with inbound event objects, and the HACEP cluster itself, which is responsible for processing said events.

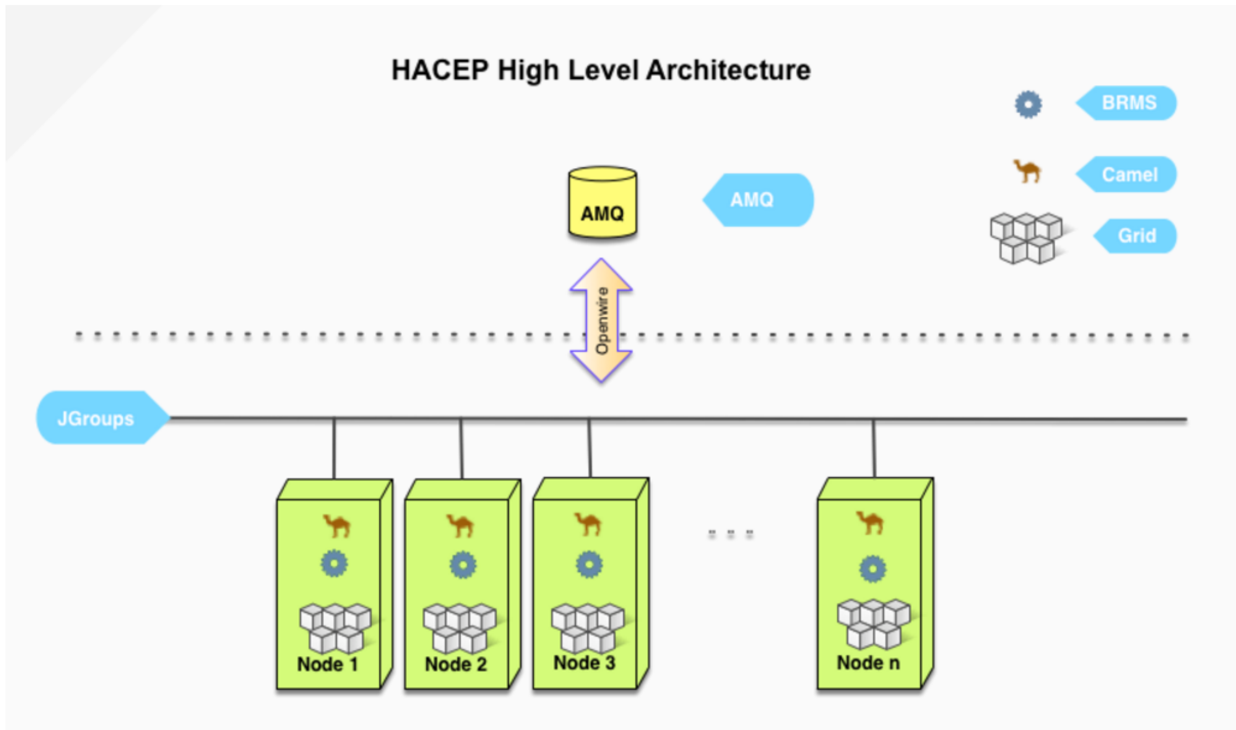


*Figure 1. High-Level HACEP Architecture*

### 2.2.1. Event Channel

The Events producer is external to the HACEP framework, however, the framework does set forward a few assumptions and recommendations about the source system. The event source must be JMS-compliant and include *JMSXGroupID* metadata with its published events. With data affinity and scaling in mind, it's highly recommended that a JMS Server like `JBoss A-MQ` be used with *message grouping* enabled. This grouping enables multiple consumers on the same queue to process, in FIFO order, messages tagged with the aforementioned JMSXGroupID. It also facilitates concurrency as multiple consumers can parallel process different message groups, each identified by the unique group ID attribute.

In use cases where the business doesn't particularly necessitate relevance of events ordering, JMS grouping could be seen as a non-viable or overly-complex option. In these cases, HACEP offers a reordering component that will internally reorder events on its nodes based on a configurable field on the events. However, it should be noted that utilizing this mechanism over JMS grouping will likely introduce some latencies due to buffering and gaps between events that must be ordered, thereby impacting overall performance.

In future versions, HACEP is slated to enable an A-MQ-backed event source to use the same grouping algorithm as JBoss Data Grid's grouping so that inbound messages are consumed directly on the group's owner node, thus furthering the framework's data affinity.

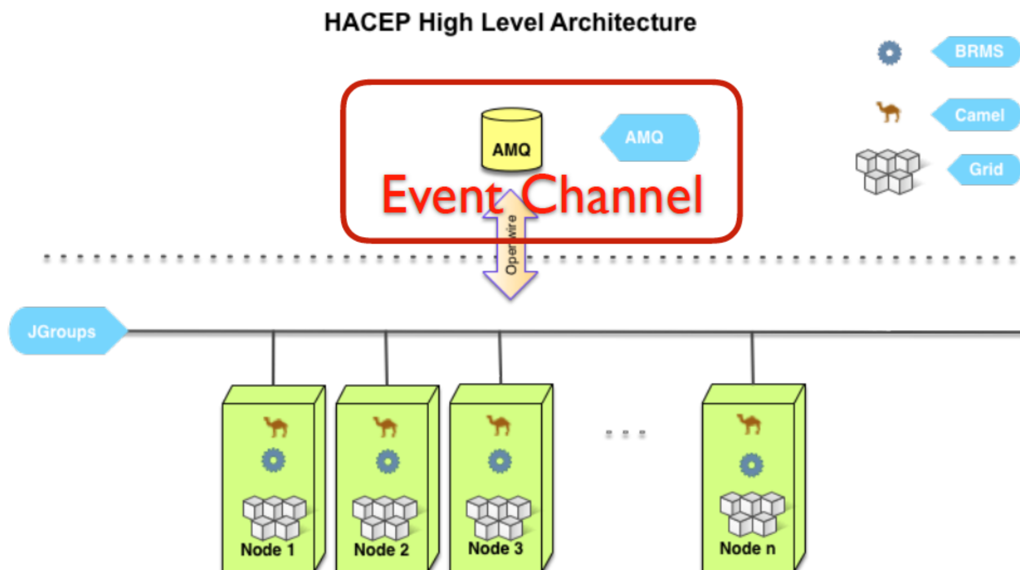


*Figure 2. Event Channel*

### 2.2.2. HACEP Node Structure

Each node within a HACEP cluster is identical. Each consists of a camel route, a portion of data relevant to its owned groups via two JDG caches, and JBoss BRMS code.

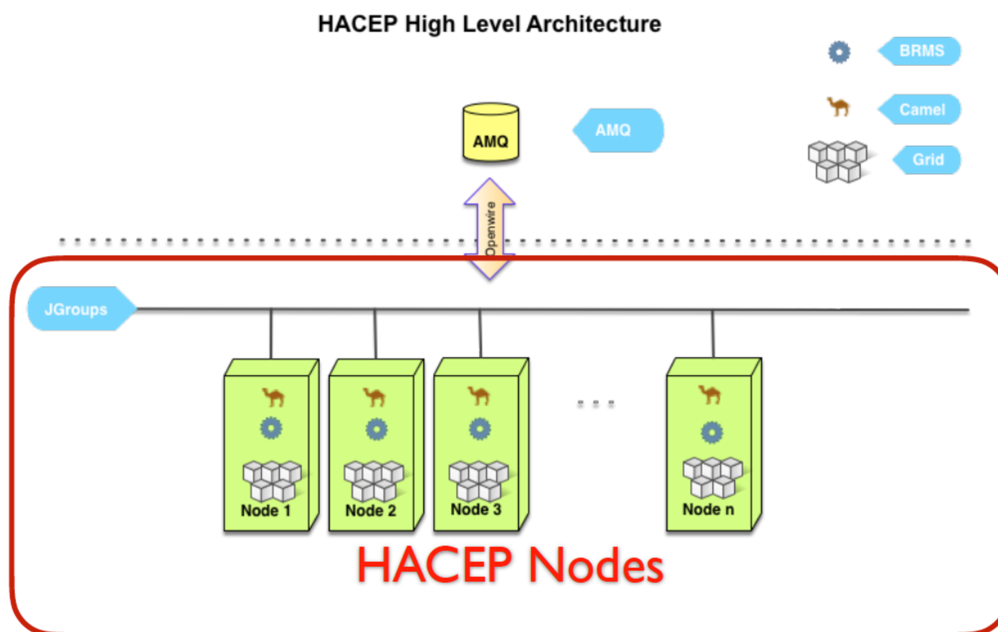


*Figure 3. HACEP Node Cluster*

### 2.2.3. Event Consumption

A HACEP cluster consists of multiple nodes, each responsible for listening to and consuming from the Event Channel event source across multiple concurrent consumers. After receiving an event, the consuming node will place the event into a JDG *Fact* (soon to be *Event* in future releases) cache. HACEP's JDG instances are configured to use its distributed topology, meaning that entries are distributed to a subset of the nodes with one of those nodes acting as the owner of the event's related group. In HACEP, this is accomplished by leveraging JDG's *grouping API*.

JBoss Data Grid's grouping API allows a group of entries to be collocated on the same node, instead of the default behavior of having each entry being stored on a node corresponding to a calculated hash code of the entry. By default JBoss Data Grid will take a hash code of each key when it is stored and map that key to a hash segment; this allows an algorithm to be used to determine the node that contains the key, allowing each node in the cluster to know the location of the key without distributing ownership information. This behavior reduces overhead and improves redundancy as the ownership information does not need to be replicated, should a node fail.

By enabling the grouping API, the hash of the key is ignored when deciding which node to store the entry on. Instead, a hash of the group is obtained and used in its place, while the hash of the key is used internally to prevent performance degradation. When the group API is in use, every node can still determine the owners of the key, which means that the group may not be manually specified. A group may either be intrinsic to the entry (generated by the key class), or extrinsic to the entry (generated by an external function).

More information can be found in the official Red Hat JBoss Data Grid Administration & Configuration Guide

Once a received event has been placed in the *event* cache, it will expire after a few milliseconds of idle time. Due to the underlying distributed topology of the grid and cache, the group owner node quickly consumes the event. There's no need to store such event facts long-term, as they're solely put into the grid to fire a synchronous notification on the node with primary ownership of the related group and later separately maintained in-cache as part of the session's respective event buffer or session snapshot via a specialized session wrapper, the *HaKieSession* object.

### 2.2.4. Session Instantiation, Resumption, and State Replication

Once an event has been received by the related group's owning node, said group owner will then retrieve the related BRMS session wrapper from the JDG *Session* cache, or if one is not found, create one for the related group. Once established, the received event is added to the wrapper object's event buffer and then injected into the BRMS session itself. Following, the session's pseudoclock is adjusted accordingly, the rules are fired, then the modified session is saved back into the JDG *Session* cache, replicating itself minimally to other non-primary nodes responsible for the group's information,

utilizing JDG's `DeltaAware` to minimize the amount of network traffic required to do so.

DeltaAware and Delta are interfaces utilized alongside JDG to allow for fine-grained replication. Rather than forcing the application to transfer a full copy of an entire object to each redundant non-primary data owner, the interfaces allow the system to identify only those parts that have been altered, send forward those changes, and allow the recipients to apply the same changes on top of the currently owned copy rather than replacing it completely. In terms of HACEP, rather than replicating the entire event buffer which exists alongside the BRMS session inside the HAKieSession wrapper object, DeltaAware can identify changes made and allow for solely the information regarding the new event information to be forwarded on to secondary nodes.

### 2.2.5. Fault Tolerance

In the event that nodes are removed or added to a HACEP cluster, the camel routes atop each node are automatically stopped. A rehashing then occurs across the cluster to rebalance session ownership and redistribute them where appropriate, according to their **Consistent Hashing**. This method of hashing also offers performance gains over traditional methods. With Consistent Hashing, the number of keys that need to be remapped on average is the *number of keys / number of slots*, as opposed to a nearly full remapping of all keys, when keys and slots are defined by a modular operation, something that is typical with more traditional implementations.

Should a failover happen to occur within the small time window in which a primary group node is mid-process firing rules, the consequential result would not necessarily reflect a valid result. In cases such as these, non-primary nodes utilize an idempotent channel to replay buffered events onto the last-known session snapshot. In cases where the event has been previously seen, the impact can be safely discarded, as it's already been captured. Eventually, the session will receive the event and resulting actions that were mid-process at the time of failure, as well as any other events that follow in the buffer. After completion, the local session has been brought up-to-date and can now be utilized as the new primary group record.

### 2.2.6. Session Snapshotting

Event buffers within the HAKieSession have a configurable limit for the number of events that can be held. Thanks to DeltaAware, this number can safely be in the thousands without severely impacting performance or network traffic, but ultimately there is a cap. While primary group owner nodes apply each event to the BRMS session on-hand when received, secondary group nodes follow a different method to preserve event sourcing. Since each event is replicated to secondary owner nodes via DeltaAware, at any given point, a secondary group node can take the BRMS session on-hand, apply its events from the buffer in-order and have an up-to-date version of the group's session ready for use, which matches that of the primary owner node (or what the primary node would contain in failover situations). Given that buffer queues have a storage limit, yet identical session capabilities must be maintained, **Session Snapshots** are utilized. When a buffer reaches the upper limit, an asynchronous thread process is initiated, which takes the current BRMS session found in local memory, applies the full queue of events within the buffer, then places the session back into storage. This process effectively records the state of the session at the end of the current queue playback. Afterwards, the

now emptied buffer can continue to capture new replicated events.

Future versions of HACEP are slated to allow a more scripted approach to configuration of snapshot usage. Whereas currently the asynchronous snapshot creation occurs at event buffer capacity, down the road configuration will allow specification of when the process should take place, be it via cron, when rules are triggered, etc. While the performance impact of the snapshot process are very minimal, such configuration will allow for even more optimized offload onto typical slow times in the system or key points throughout the business process.

# 3. Reference Architecture Environment

## 3.1. Overview

This reference architecture takes advantage of the provided `JBoss EAP 7` installer script to establish multiple basic application servers for hosting the `HACEP` framework and its various components. External to the EAP instances, a `JBoss A-MQ` cluster is established as the Event Channel, or origin point of those events which will feed the HACEP system. A separate `ZooKeeper` ensemble is used for LevelDB replication pertaining to the A-MQ cluster.

## 3.2. JBoss EAP 7.0.1

`Red Hat JBoss Enterprise Application Platform 7.0.1`, Red Hat's Java EE-based application server runtime, is utilized as the container for deployment of the HACEP framework. Various capabilities provided by the container are leveraged as part of a HACEP enterprise-level integration, such as `Weld` Contexts and Dependency Injection, `JGroups` integration for ease of node clustering and communication, resource adapters for simplified communication with external `JMS` brokers, and more.

## 3.3. HACEP Cluster Nodes

The HACEP framework is presented as a series of Maven dependencies, which can be integrated within a new or existing application to facilitate the establishment of a highly-available and scalable Complex Event Processing application or service. While there are various extracurricular modules provided as part of the HACEP source code worth mentioning later, the core parts required for integration consist of the `hacep-core`, `hacep-core-model`, and `hacep-core-camel` modules.

At a high level, the `hacep-core` module provides the majority of the HACEP code, including various extendable configuration interfaces, data grid cache management & optimization, JGroups clustering orchestration, and BRMS session management.

The `hacep-core-model` module is responsible for hosting integral POJO definitions used throughout the various modules, such as the **Fact** object, which is related to rule sessions, and the *SessionKey*, which correlates to data grid cache indexing.

The `hacep-core-camel` module serves to configure the coordination point for interacting with the external JMS **Event Channel** component, consuming events from the origin point and kicking off the HACEP processing of the event input.

# 3.4. HACEP Core Components

The following technologies serve integral roles within the HACEP framework core code.

## 3.4.1. JGroups

In order to provide scalability, HACEP core code leverages JGroups for multicast communication between nodes. From a clustering perspective, JGroups provides leadership election among nodes as well as notification of addition and removal of nodes which allows the core code to properly implement fault tolerance and distribution of responsibilities among the various members in the cluster. JGroups is leveraged internally by the data grid for cross-network communication in HACEP's distributed topology.

## 3.4.2. Routing

As mentioned previously, camel routes are used as a means of communicating with the Event Channel, serving as the inbound origin point of rule-related events. The core HACEP code establishes a route with a JMS endpoint from which a configurable amount of concurrent consumers will take in events from the channel and relay them to the in-memory data grid.

In cases where nodes are added or taken away from a HACEP cluster, the internal camel configuration allows for pause of route functionality while session node ownership and distribution is dealt with internally before resuming operations.

## 3.4.3. JBoss BRMS

The HACEP framework was born out of a desire to leverage the complex event processing (CEP) capabilities of the JBoss Business Rule Management System (BRMS) in such a way as to meet the expectations and requirements of enterprise-level applications. While the rules engine itself lends to a rather large breadth of CEP functionality, in and of itself worthy of significant documentation, it offers nothing out-of-the-box which assists in high-availability or scalability. At the heart of HACEP, events from the inbound Event Channel are fed to distributed, replicated, CEP-capable BRMS rules sessions, where they are processed in order to facilitate some decision process defined by rules, typically authored by application developers, business users, or stake owners.

## 3.4.4. Data Grid

The distributed in-memory key/value data store product is used to facilitate cross-node replication of key objects such as inbound event facts and rule sessions. With HACEP, Infinispan as delivered by JDG is used in a distributed topology with the Grouping API enabled, allowing "ownership" of sessions by HACEP nodes in relation to data affinity. HACEP uses two caches called "event" (for holding *facts*) and "session" within the grid. The data grid internally uses JGroups for cross-node multicast communication.

## 3.5. Event Channel Components

The Event Channel origin point used within this reference architecture is comprised of the following technologies.

### 3.5.1. JBoss A-MQ 6.2.1 Cluster

Currently, HACEP core code only requires that an Event Channel server be JMS-compliant and include the JMSXGroupId attribute on all inbound events. The former ensures assumed Camel routing functionality is possible, while the latter is used for node ownership identification. In this reference architecture, a JBoss A-MQ cluster is used for the Event Channel for both the example applications included in the HACEP source code as well as for the example integration application detailed in later chapters.

In future HACEP versions, optimizations are planned to use A-MQ-based systems exclusively for the Event Channel so that the grouping algorithm used for message delivery can be mated with the one utilized by the internal data grid, making direct delivery of events to relevant nodes possible and furthering data affinity.

### 3.5.2. ZooKeeper Cluster

The JBoss A-MQ cluster used within this reference architecture relies on a ZooKeeper-based replicated LevelDB Store for master/slave coordination and redundancy. Note that this is only one of various replication strategies provided by A-MQ. Officially as of the time of writing, support within the A-MQ cluster for this specific LevelDB strategy is included as technical preview only, but given that the Event Channel configuration largely stands independently of HACEP core code, the replicated LevelDB store is utilized here as a safe example configuration which can easily be swapped to any of the other provided solutions at the behest of application and environment requirements and consistent with enterprise support.

# 4. Creating the Environment

## 4.1. Prerequisites

This reference architecture assumes a supported platform (Operating System and JDK). For further information, refer to the Red Hat Supported Configurations for EAP 7.

As mentioned before, JBoss A-MQ is used as the Event Channel origin point herein. The following documentation encompasses setup for this choice. Other JMS servers are viable options at this time, though likely to be phased out in future HACEP versions. Should you choose to use an alternative, please refer to the product's installation guide in lieu of Event Channel setup for your stack.

## 4.2. Downloads

This document makes reference to and use of several files included as an accompanying attachment to this document. Future versions of the framework can be found at Red Hat Italy's GitHub project page, however, the attached files can be used as a reference point during the integration process and include a copy of the application developed herein, as well as a copy of the HACEP framework source code as of time of writing, version SNAPSHOT-1.0, which best pairs with the content of the paper:

https://access.redhat.com/node/2542881/40/0

If you do not have access to the Red Hat customer portal, please see the Comments and Feedback section to contact us for alternative methods of access to these files.

In addition to the accompanying attachment, download prerequisites, including:

- Red Hat JBoss Enterprise Application Platform 7.0.0 Installer
- Red Hat JBoss Enterprise Application Platform 7.0 Update 01
- Red Hat JBoss A-MQ 6.2.1
- ZooKeeper 3.4.8
- Red Hat Maven repository activemq-rar.rar

## 4.3. Installation

### 4.3.1. ZooKeeper Ensemble

After downloading the ZooKeeper release archive, choose a location where it will reside. It is possible to utilize a single server node to host both A-MQ and ZooKeeper. However, in order to facilitate similarities to likely production environments, this document assumes separate server clusters for each technology. The root installation location used herein for ZooKeeper is */opt/zookeeper_3.4.8/*.

Heap memory available to each ZooKeeper instance can be configured via creating a file in the *conf* directory called *java.env* containing the following text:

```
export JVMFLAGS="-Xms2G -Xmx3G"
```

The data directory and port of usage utilized by ZooKeeper are likewise specified in a file within the *conf* directory called *zoo.cfg*:

***Listing 1. zoo.cfg***

```
clientPort=2181
dataDir=/opt/zookeeper_3.4.8/data
syncLimit=5
tickTime=2000
initLimit=10
dataLogDir=/opt/zookeeper_3.4.8/data
server.1=zk1:2888:3888
server.2=zk2:2888:3888
server.3=zk3:2888:3888
```

The minimum number of nodes suggested for a ZooKeeper ensemble is three instances, thus the server entries seen above, where 'zk[1-3]' are host entries aliasing the local network IP addresses of each ZooKeeper server. Each node within an ensemble should use an identical copy of the *zoo.cfg* file. Note the ports 2888 and 3888 given above, which in addition to the clientPort of 2181, allow for quorum and leader elections and thus may require firewall configuration on your part to ensure availability. If you desire to run multiple ZooKeeper instances on a single machine, rather than on three separate servers as done herein, each server instance should have a unique assignment of these two ports (2888:3888, 2889:3889, 2890:3890, etc).

Sample copies of both *java.env* and *zoo.cfg* have been included in the accompanying attachment file under the */config_files/zookeeper/* directory.

Once configured, the ZooKeeper instance can be started via the *[install_dir]/bin* directory with the following command:

```
# ./zkServer.sh start
```

You can tail the instance's log file with the following command:

```
# tail -f [root_install_dir]/zookeeper.out
```

For further information on configuring, running, and administering ZooKeeper instances and ensembles, please refer to the Apache Zookeeper Getting Started Guide and Administrator's Guide.

### 4.3.2. JBoss A-MQ Cluster

After downloading Red Hat JBoss A-MQ 6.2.1, choose a location where it will reside. The root installation location used herein for A-MQ is */opt/amq_6.2.1/*.

Prior to starting the instance, further configuration is required for utilizing the replicated LevelDB store strategy. Should you choose to use a different persistence strategy, now is the time to configure it as detailed in the Red Hat JBoss A-MQ Guide to Configuring Broker Persistence. To follow along with the strategy adhered to within, proceed as follows:

First step is to ensure that your system has at least one user set up. Open the file *[root_install]/etc/users.properties* and ensure that at least an admin user is present and not commented out:

```
# You must have at least one users to be able to access JBoss A-MQ resources

*admin=admin,admin,manager,viewer,Operator, Maintainer, Deployer, Auditor, Administrator,
SuperUser*
```

If you would like to verify the basic installation of A-MQ at this time, steps for doing so can be found in the Red Hat A-MQ Installation Guide under the heading *Verifying the Installation*.

Next, configure the persistence mechanism by editing the *[root_install]/etc/activemq.xml* file to include the <*persistenceAdapter*> section as shown below:

```
<broker brokerName="broker" persistent="true" ... >
  ...
    <persistenceAdapter>
      <replicatedLevelDB
      directory="/opt/amq_6.2.1/data"
      replicas="3"
      bind="tcp://192.168.2.210:61619"
      hostname="amq1"
      zkAddress="zk1:2181,zk2:2181,zk3:2181"
      zkPassword="password"
      zkPath="/opt/zookeeper_3.4.8/activemq/leveldb-stores" />
    </persistenceAdapter>
  ...
</broker>
```

Note that the binding is unique to the network interface IP address of the specific node, while the zkAddress will be identical across all A-MQ instances. The number of replicas should reflect the number of nodes within your ZooKeeper ensemble.

The last step in configuring the A-MQ instance involves editing the *[root_install]/bin/setenv* script file. Inside, you can specify the amount of Heap space available to the instance, as well as exporting a JVM arg called *SERIALIZABLE_PACKAGES*:

```
#!/bin/sh
...
if [ "$KARAF_SCRIPT" = "start" ]; then
    JAVA_MIN_MEM=2G
    JAVA_MAX_MEM=3G
    export JAVA_MIN_MEM # Minimum memory for the JVM
    export JAVA_MAX_MEM # Maximum memory for the JVM
fi
...
export KARAF_OPTS="-Dorg.apache.activemq.SERIALIZABLE_PACKAGES=\"*\""
```

The *SERIALIZABLE_PACKAGES* is related to HACEP's usage of the ObjectMessage, which depends on Java [de]serialization to [un]marshal object payloads. Since this is generally considered unsafe, given that payloads could exploit the host system, the above option can be tuned to be more specific to those object types which we know the system will be dealing with. For more details on ObjectMessage, refer to the relevant ActiveMQ documentation. A more finely-tuned example would read as follows:

```
export KARAF_OPTS="-
Dorg.apache.activemq.SERIALIZABLE_PACKAGES=\"it.redhat.hacep,com.redhat.refarch.hacep,you
r.company.model.package\""
```

After configuration is complete, you can start the A-MQ instance by issuing the following command from the *[root_install]/bin/* directory:

```
# ./start
```

You can tail the instance's log file with the following command:

```
# tail -f [root_install_dir]/data/log/amq.log
```

At this point, your replicated A-MQ Event Channel is now fully established and running, ready to feed events to the HACEP framework integration application.

### 4.3.3. EAP 7.0.1

Given that HACEP requires, at a minimum, two clustered instances, EAP installations in a production environment will likely result in use of domain clustering functionality. However, given the documentation necessary to effectively cover active/passive clustering appropriate for a production

environment, the reader is instead encouraged to utilize the Red Hat Reference Architecture for Configuring a JBoss 7 EAP Cluster when establishing production-ready environments. In order to maintain focus on the integration and functionality of the HACEP framework in a proof-of-concept environment, two separate EAP standalone server instances are used herein, allowing JGroups to do the inter-node communication necessary for HACEP to function.

### Installer Script

After downloading the EAP 7 installer, execute it using the following command within the directory containing the .jar file:

```
java -jar jboss-eap-7.0.0-installer.jar
```

You will be presented with various options aimed at configuring your server instance, each followed by an opportunity to confirm your selection. The following choices should be used when installing each EAP instance:

- *language*: enter to select default for English or make a choice appropriate to your needs
- *license*: after reading the the End User License Agreement, enter 1 to continue
- *installation path*: /opt/EAP_7/
- *pack installation*: enter 0 to accept the default selection of packs to be installed
- *administrative user name*: enter to accept the default of *admin*
- *administrative user password*: enter *password!* twice to confirm the entry
- *runtime environment configuration*: no additional options required, enter to accept the default of 0
- *automatic installation script*: though not necessary, if you'd like to generate an installation script for the settings chosen, you can do so at this time

After completion, you should receive a confirmation message that the installation is done:

```
[ Console installation done ]
```

### Patch Application

With basic installation complete, start the server in standalone mode in order to apply the 7.0.1 patch file by issuing the following command:

```
[root_install_dir]/bin/standalone.sh -b 0.0.0.0 -bmanagement 0.0.0.0
```

Note that in the lines above, an IP binding of *0.0.0.0* is provided for both the server & management ports of the address, which in most environments allows the server to listen on all available interfaces.

If you've chosen to run installations across various virtual machines, you may also find these parameters necessary to allow configuring the server away from *localhost*.

Once the server has started, you can access the management UI interface at *http://[ip_address]:9990* with the credentials supplied during installation. From the landing page, you can then select **Apply a Patch** in the bottom-left to start the patching process. Provide the downloaded patch file when requested, then complete the process. Following completion, stop the standalone server.

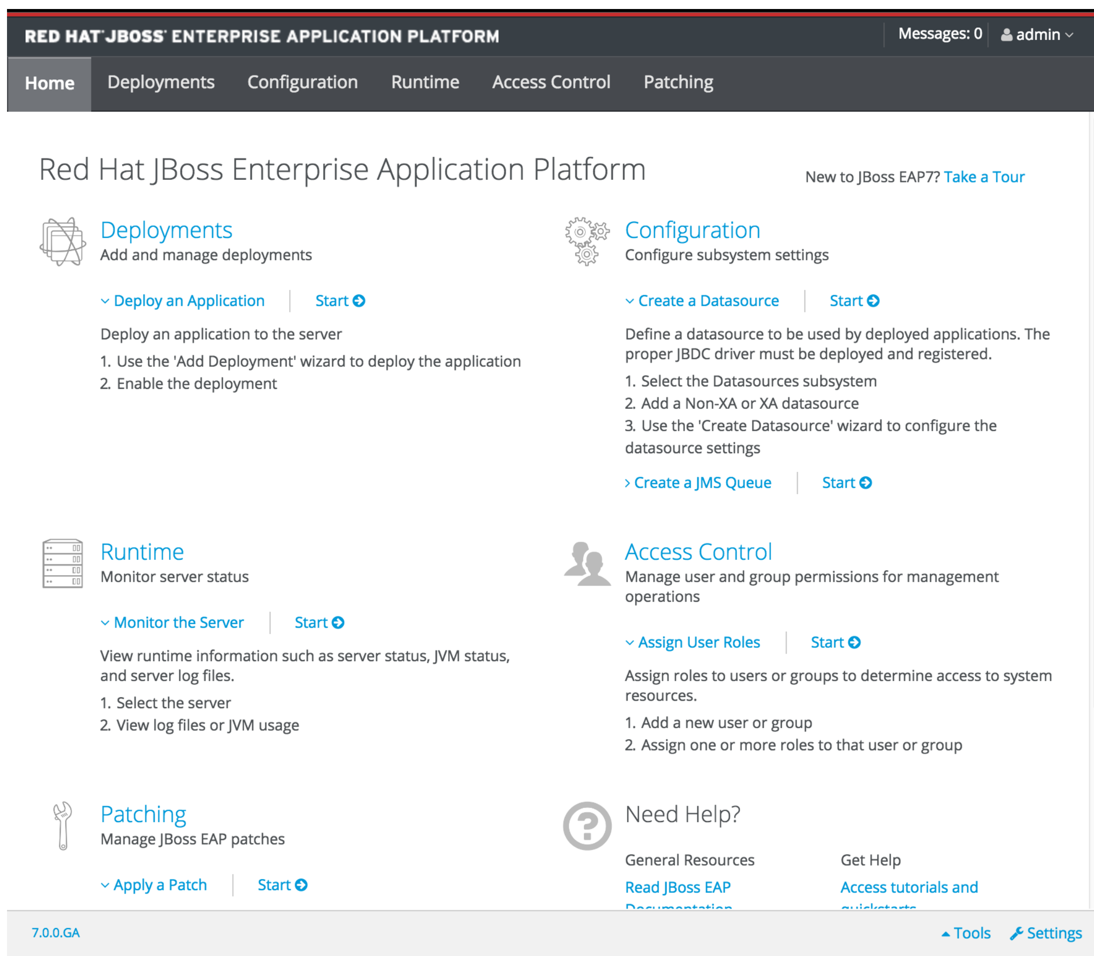


***Figure 4. EAP Management UI Landing Page***

## Resource Adapter Configuration

In order to leverage EAP's resource adapters to connect and interact with the Event Channel cluster, further configuration is required. While the basic steps are outlined below, should you encounter issues or desire further description of what the process entails, reference the Red Hat JBoss AM-Q Guide for Integrating with JBoss Enterprise Application Platform.

The first step is to edit the file *[root_install]/standalone/configuration/standalone.xml* as follows. Near the top of the document, add the *messaging-activemq* extension entry just as the other extensions are enabled as shown below. Doing so enables the ActiveMQ Artemis messaging system embedded within EAP which can be used as an internal messaging broker. However, since we've chosen to utilize an external broker for the Event Channel, we're simply leveraging the inclusion of various Java EE API dependencies required by the resource adapter we'll be using by enabling this extension.

```
<extension module="org.wildfly.extension.messaging-activemq"/>
```

Further down the file, amongst the other subsystem entries, supply an empty entry for the *messaging-activemq* system as shown in bold below:

```
<subsystem xmlns="urn:jboss:domain:mail:2.0">
   <mail-session name="default" jndi-name="java:jboss/mail/Default">
      <smtp-server outbound-socket-binding-ref="mail-smtp"/>
   </mail-session>
</subsystem>
<subsystem xmlns="urn:jboss:domain:messaging-activemq:1.0"/>
<subsystem xmlns="urn:jboss:domain:naming:2.0">
   <remote-naming/>
</subsystem>
```

Next, locate the *ejb3* subsystem entry and add an *mdb* entry for the connection pool that we'll be using as shown in bold below:

```
<subsystem xmlns="urn:jboss:domain:ejb3:4.0">
   <session-bean>
      <stateless>
         <bean-instance-pool-ref pool-name="slsb-strict-max-pool"/>
      </stateless>
      <stateful default-access-timeout="5000" cache-ref="simple" + passivation-disabled-cache-
ref="simple"/>
      <singleton default-access-timeout="5000"/>
   </session-bean>
   <mdb>
      <resource-adapter-ref resource-adapter-name="activemq-rar.rar"/>
      <bean-instance-pool-ref pool-name="mdb-strict-max-pool"/>
   </mdb>
   <pools>
      <bean-instance-pools>
         <strict-max-pool name="slsb-strict-max-pool" derive-size="from-worker-pools" instance-
acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
         <strict-max-pool name="mdb-strict-max-pool" derive-size="from-cpu-count" instance-
acquisition-timeout="5" instance-acquisition-timeout-unit="MINUTES"/>
      </bean-instance-pools>
   </pools>
```

Finally, add an entry to the resource-adapters subsystem section detailing the system we're utilizing to communicate with the external Event Channel cluster as shown below:

```
<subsystem xmlns="urn:jboss:domain:resource-adapters:4.0">
 <resource-adapters>
  <resource-adapter id="activemq-rar.rar">
   <archive>
    activemq-rar.rar
   </archive>
   <transaction-support>XATransaction</transaction-support>
   <config-property name="ServerUrl">
     failover:(tcp://amq1:61616,tcp://amq2:61616,tcp://amq3:61616)?jms.rmIdFromConnectio
nId=true&maxReconnectAttempts=0
   </config-property>
   <config-property name="UserName">
    admin
   </config-property>
   <config-property name="Password">
    admin
   </config-property>
   <config-property name="UseInboundSession">
    true
   </config-property>
   <connection-definitions>
    <connection-definition
      class-name="org.apache.activemq.ra.ActiveMQManagedConnectionFactory"
      jndi-name="java:/HacepConnectionFactory"
      enabled="true"
      pool-name="ConnectionFactory">
     <config-property name="UseInboundSession">
      false
     </config-property>
     <xa-pool>
      <min-pool-size>1</min-pool-size>
      <max-pool-size>20</max-pool-size>
      <prefill>false</prefill>
      <is-same-rm-override>false</is-same-rm-override>
     </xa-pool>
```

```
          <recovery>
            <recover-credential>
              <user-name>admin</user-name>
              <password>admin</password>
            </recover-credential>
          </recovery>
        </connection-definition>
      </connection-definitions>
      <admin-objects>
        <admin-object class-name="org.apache.activemq.command.ActiveMQQueue"
          jndi-name="java:/queue/hacep-facts"
          use-java-context="true"
          pool-name="hacep-facts">
          <config-property name="PhysicalName">
            hacep-facts
          </config-property>
        </admin-object>
      </admin-objects>
    </resource-adapter>
  </resource-adapters>
</subsystem>
```

Note the ServerUrl value, which takes the form of the *failover:* protocol, allowing us to use the full A-MQ cluster in a fault-tolerant way. The username, password, and URL should be customized to fit your requirements. The *&* is also intentional as to escape the character to survive the system's translation of the configuration resource. Further information about the values provided and changes made for working with a cluster can be found in the Red Hat JBoss AM-Q Guide for Integrating with JBoss Enterprise Application Platform.

The last step of configuring the needed resource adapter is to provide the archive file which was indicated in the last configuration step above. The file can be downloaded from the Red Hat Maven repository. Copy the *activemq-rar.rar* file into the *[root_install]/standalone/deployments/* directory, then start the standalone server again. If everything was configured correctly, you should see information regarding the newly established JNDI connections as part of the server startup output:

```
13:11:43,390 INFO  [org.jboss.as.connector.deployment] (MSC service thread 1-1)
WFLYJCA0002: Bound JCA AdminObject [java:/queue/hacep-facts]

13:11:43,390 INFO  [org.jboss.as.connector.deployment] (MSC service thread 1-1)
WFLYJCA0002: Bound JCA ConnectionFactory [java:/HacepConnectionFactory]

13:11:43,430 INFO  [org.jboss.as.server] (ServerService Thread Pool -- 35) WFLYSRV0010:
Deployed "activemq-rar.rar" (runtime-name : "activemq-rar.rar")
```

### Environment Configuration

Lastly, to ensure that the JVM has ample heap space available, JGroups has the configuration needed, and the resource adapter knows how to deal with the ActiveMQ ObjectMessage caveat seen before, we need to edit the *[root_install]/bin/standalone.conf* file as shown below. Note that the *SERIALIZABLE_PACKAGES* option can be tailored accordingly as previously described.

```
 ...
#
# Specify options to pass to the Java VM.
#
if [ "x$JAVA_OPTS" = "x" ]; then
   JAVA_OPTS="-Xms2G -Xmx3G -XX:MetaspaceSize=128M -XX:MaxMetaspaceSize=256m
      -Djava.net.preferIPv4Stack=true"
   JAVA_OPTS="$JAVA_OPTS
      -Djboss.modules.system.pkgs=$JBOSS_MODULES_SYSTEM_PKGS
      -Djava.awt.headless=true"
else
   echo "JAVA_OPTS already set in environment; overriding default settings with values:
   $JAVA_OPTS"
fi
 ...
JAVA_OPTS="$JAVA_OPTS -Djgroups.tcp.address=192.168.2.200
-Dorg.apache.activemq.SERIALIZABLE_PACKAGES=\"*\""
```

## 4.4. Conclusion

At this point, your environment has been fully established, configured, and made ready for deployment of any application integrating the HACEP framework. Assuming your ZooKeeper ensemble and A-MQ cluster are running and functional, you should now be ready to take advantage of the JNDI connection to the Event Channel via standard Resource lookup:

```
@Resource(lookup = "java:/HacepConnectionFactory")
private ConnectionFactory connectionFactory;
```

# 5. Design and Development

## 5.1. HACEP Integration and Source Examples

This reference architecture includes an example application that is designed, deployed, and tested herein. The application consists of four modules: the integration code with RESTful interface, the supporting data model, the rules package, and a runner script used to populate the Event Channel and exercise the application.

This application is alternatively referred to as *hacep-integration* and the source code is available as part of the downloaded attachment accompanying this paper. While a complete copy of the example application is provided with this reference architecture, this section walks the reader through every step of design and development. By following the steps outlined in this section, the reader is able to replicate the original effort and recreate every component of the application.

The HACEP source code itself contains two examples of HACEP integration, one intended for usage via command line, and the other being an example EAP integration itself. While the structure of the latter was referenced quite heavily in the creation of the integration application detailed below, the two are somewhat different in that the provided source example ties into a larger subset of modules intended for demonstration and exhibit of the HACEP framework and its features, whereas the application built herein minimally shows the efforts required for integration into your own application or service code and allows coverage of the steps from start to finish without the need to filter out extraneous parts or code.

## 5.2. Project Setup

This reference architecture assumes that the previous installation and configuration steps have been followed and the environment set up. It's also assumed that the HACEP source code has been copied out of the accompanying download for reference. In order to support the JBoss Maven repositories necessary for working with the project, modify or create a *settings.xml* file in your local machine's */.m2/* directory, typically located in the user's home directory. An example *settings.xml* is included in the HACEP source code under *example-maven-settings* and instructions for where to download and set up the mentioned repositories can be found in the project README within the source code.

## 5.3. Running the CLI HACEP Example

Within the source code in the *hacep-examples* directory, resides a *hacep-playground* example which can be used to quickly preview the functionality of the HACEP framework and become familiar with the various commands present within the more complex *hacep-eap-playground* example. In order to run a four-node example from one machine, you can run the following commands in separate terminals, then take the various instances on and offline to see the clustering in action.

```
mvn -P run -DnodeName=node1 -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=localhost
-Dgrid.buffer=5000 -Dqueue.url=tcp://localhost:61616 -Dqueue.security=true
-Dqueue.usr=admin -Dqueue.pwd=admin

mvn -P run -DnodeName=node2 -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=localhost
-Dgrid.buffer=5000 -Dqueue.url=tcp://localhost:61616 -Dqueue.security=true
-Dqueue.usr=admin -Dqueue.pwd=admin

mvn -P run -DnodeName=node3 -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=localhost
-Dgrid.buffer=5000 -Dqueue.url=tcp://localhost:61616 -Dqueue.security=true
-Dqueue.usr=admin -Dqueue.pwd=admin

mvn -P run -DnodeName=node4 -Djava.net.preferIPv4Stack=true -Djgroups.bind_addr=localhost
-Dgrid.buffer=5000 -Dqueue.url=tcp://localhost:61616 -Dqueue.security=true
-Dqueue.usr=admin -Dqueue.pwd=admin
```

Once running, each instance will give you access the the HACEP CLI prompt. To start, use the *help* command to view the various commands and usage available. For example, *info* will show you a small amount of information about the active HACEP cluster, as seen below:

```
----------------------------------------
                HACEP CLI
----------------------------------------

> help
Commands:
address
        Address of this cluster node
all <cache>
        List all values.
change <user>
        change <user> <oldpassword> to <newpassword>
get <cache> <key> <player>
        Get an object from the <cache>.
help
        List of commands.
info [<cache>]
        General information or specific information on cache.
local <cache>
        List all local valuesFromKeys.
login <user> <password>
        Login the <user>
logout <user>
        Logout the <user>
primary <cache>
        List all local valuesFromKeys for which this node is primary.
quit|exit|q|x
        Exit the shell.
replica <cache>
        List all local values for which this node is a replica.
> info
Cache Manager Status: RUNNING
Cache Manager Address: lab1-55621
Coordinator address: lab1-20909
Is Coordinator: false
Cluster Name: HACEP
Cluster Size: 2
Member list: [lab1-20909, lab1-55621]
Caches: [fact, session]
```

# 5.4. Running the EAP HACEP Example

The other provided example application included with the HACEP source code is one that demonstrates EAP integration. Since we already have an environment established for building our own integration, testing this example is as simple as building the *.war* file via *mvn clean package* in the root directory of the HACEP codebase. Next, modify the JNDI name within the *standalone.xml* resource adapter definition to read *java:/HACEPConnectionFactory* rather than *java:/HacepConnectionFactory* (notice the capitalization differences). Following, copy the .war file out of the *hacep-eap-playground/target* directory into the *[root_eap_install]/standalone/deployments/* directory on the intended HACEP nodes and start the standalone server.

Once up and running, a similar set of commands is available for use via a RESTful interface. The format for accessing them via browser is *[ip_addr]:8080/[war_name]/execute/info*, such as seen below:

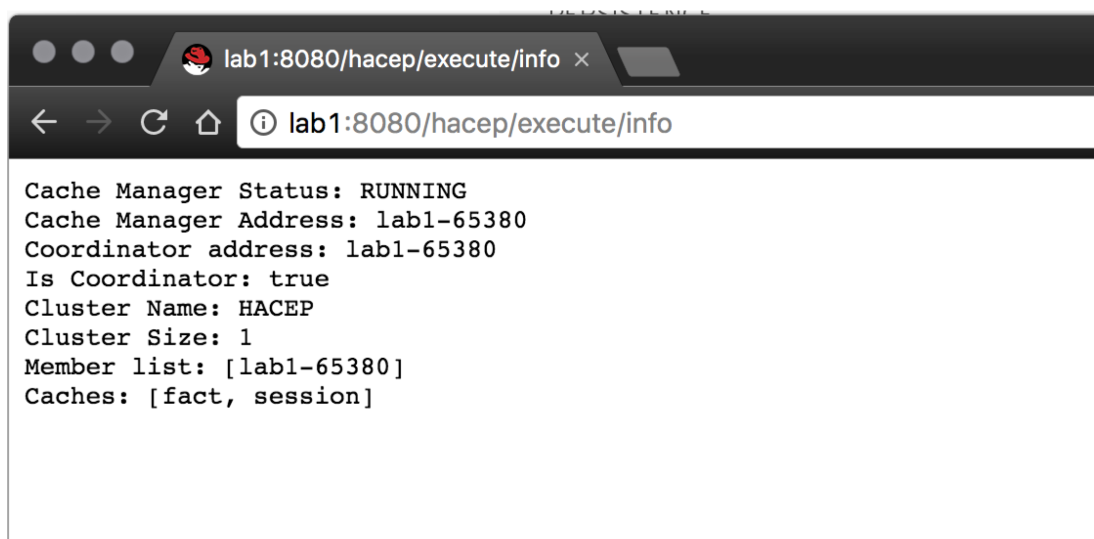


*Figure 5. EAP Playground Example Info Screen*

The HACEP source code also includes a module called *hacep-perf-client* that contains a runnable Java class for populating the Event Channel, lending the ability to monitor the HACEP nodes while in action. While the servers are running, navigate to the perf-client's target directory, then execute the script using the following format:

```
java -Dduration=480 -Dconcurrent.players=10 -Ddelay.range=5 -Devent.interval=1
-Dtest.preload=true -Dtest.messages=20000 -Dbroker.host="[AMQ_MASTER_NODE_IP]:61616"
-Dbroker.authentication=true -Dbroker.usr=admin -Dbroker.pwd=admin
-Dorg.apache.activemq.SERIALIZABLE_PACKAGES="*" -cp hacep-perf-client-1.0-SNAPSHOT.jar
it.redhat.hacep.client.App
```

While executing, a multitude of events are published to the Event Channel by parallel threads and thus consumed by the various HACEP nodes, potentially triggering rules regarding a video game POJO model included within the source code. These actions and rules should lead to some output within the server logs. While running, it's also possible to take a HACEP node offline and back on again to witness

the logs of the rebalancing process in action. When satisfied with the demonstration, stop the jar process, then un-deploy the EAP Playground and reset the JNDI reference name within standalone.xml as it was prior to setting up this example.

## 5.5. Integrating HACEP into an Application

The example HACEP integration application built herein will be comprised of four modules. The first, the *integration-app* module, will be the main part of the application extending various HACEP configurations and housing the RESTful interface we'll use to monitor the HACEP instances. The second module, *integration-model*, will house the POJO models that will be used in both the main code and the rules module. Next, the *integration-rules* module will hold the BRMS rules and *KIE* configuration used to establish the CEP sessions. Lastly, a module called *purchase-publisher* is included that will act as a runner application to feed events into the Event Channel in order to exercise our HACEP-infused application.

### 5.5.1. Parent Project Configuration

The parent project which will house the various modules mentioned in the previous section is called *hacep-integration*. This parent project will do little more than orchestrate the dependencies of the inner modules and control packaging and various other maven commands. We start by initializing a new Maven Project in JBoss Developer Studio with the following information and hitting Finish:

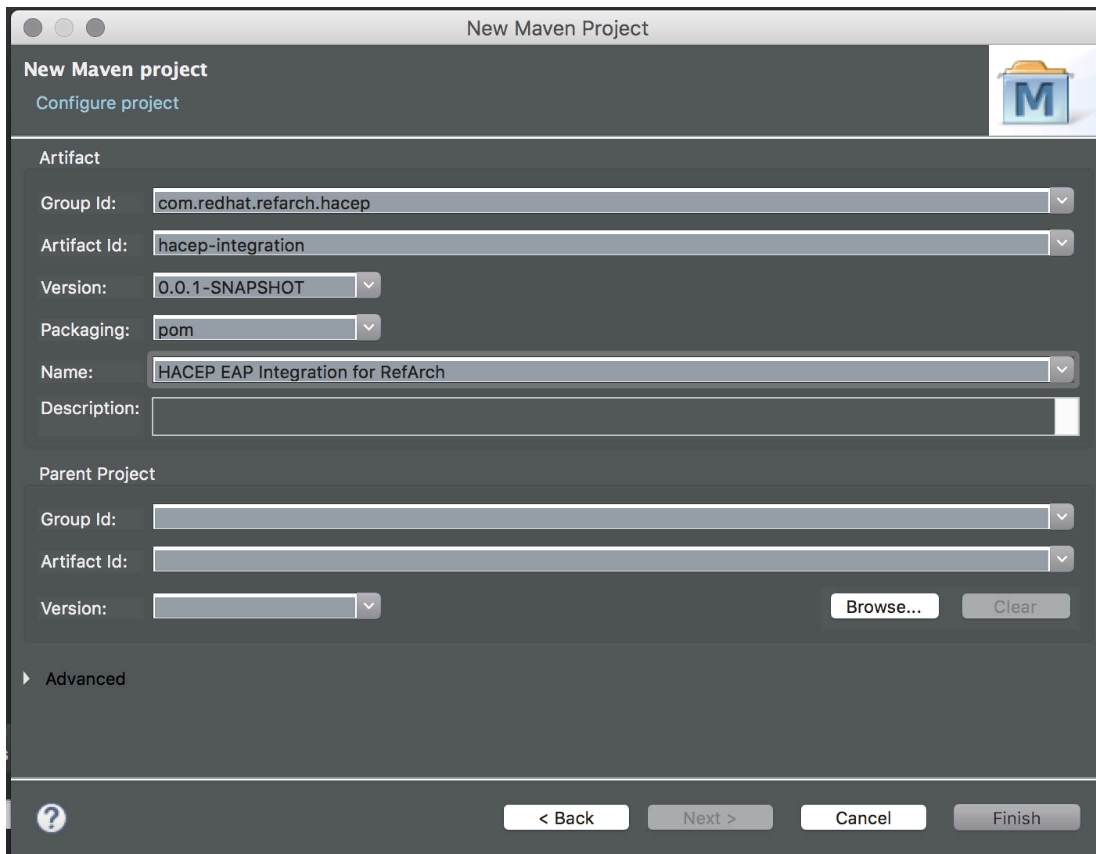


*Figure 6. Parent Pom Information*

With the outlying parent project now present, we can edit the *pom.xml* file to lay out the project structure required. Begin by adding the modules, license, and properties to the pom file:

```
<modules>
    <module>integration-app</module>
    <module>integration-rules</module>
    <module>integration-model</module>
    <module>purchase-publisher</module>
</modules>

<licenses>
    <license>
        <name>Apache License, Version 2.0</name>
        <distribution>repo</distribution>
        <url>www.apache.org/licenses/LICENSE-2.0.html</url>
    </license>
</licenses>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.plugin.version>3.3</maven.compiler.plugin.version>
    <maven.resources.plugin.version>2.6</maven.resources.plugin.version>
    <maven.war.plugin.version>2.6</maven.war.plugin.version>
    <maven.jar.plugin.version>2.6</maven.jar.plugin.version>
    <maven.surefire.plugin.version>2.19.1</maven.surefire.plugin.version>
    <maven.dependency.plugin.version>2.10</maven.dependency.plugin.version>
    <maven.exec.plugin.version>1.3.2</maven.exec.plugin.version>

    <hacep.version>1.0-SNAPSHOT</hacep.version>
    <junit.version>4.12</junit.version>
    <mockito.version>1.10.19</mockito.version>
    <log4j.version>1.2.17</log4j.version>
    <slf4j.version>1.7.12</slf4j.version>
</properties>
```

These sections define the child modules that will be included, the Apache license which the software will fall under, and various version variables we'll need throughout the *pom* configuration. Next, add a default profile referencing the JBoss repositories previously mentioned:

```
<profiles>
  <profile>
  <id>supported-GA</id>
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  <properties>
    <version.org.infinispan>8.3.0.Final-redhat-1</version.org.infinispan>
    <version.org.jboss.fuse>6.2.1.redhat-084</version.org.jboss.fuse>
    <version.org.jboss.bom.brms>6.3.0.GA-redhat-3</version.org.jboss.bom.brms>
    <version.org.jboss.bom.eap>6.4.5.GA</version.org.jboss.bom.eap>
  </properties>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.bom.eap</groupId>
        <artifactId>jboss-javaee-6.0-with-transactions</artifactId>
        <version>${version.org.jboss.bom.eap}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

      <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-bom</artifactId>
        <version>${version.org.infinispan}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>

      <dependency>
        <groupId>org.jboss.bom.brms</groupId>
        <artifactId>jboss-brms-bpmsuite-bom</artifactId>
        <version>${version.org.jboss.bom.brms}</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
```

```
        <dependency>
          <groupId>org.jboss.fuse.bom</groupId>
          <artifactId>jboss-fuse-parent</artifactId>
          <version>${version.org.jboss.fuse}</version>
          <type>pom</type>
          <scope>import</scope>
          <exclusions>
            <exclusion>
              <groupId>com.google.guava</groupId>
              <artifactId>guava</artifactId>
            </exclusion>
          </exclusions>
        </dependency>
      </dependencies>
    </dependencyManagement>
  </profile>
</profiles>
```

Beneath the profiles, add the *dependencyManagement* and *dependencies* sections which will orchestrate child module dependency versions, inclusions, and Java EE API usage:

```
<dependencyManagement>
    <dependencies>

        <!-- HACEP Framework Dependencies -→
        <dependency>
            <groupId>it.redhat.jdg</groupId>
            <artifactId>hacep-core</artifactId>
            <version>${hacep.version}</version>
        </dependency>
        <dependency>
            <groupId>it.redhat.jdg</groupId>
            <artifactId>hacep-core-model</artifactId>
            <version>${hacep.version}</version>
        </dependency>
        <dependency>
            <groupId>it.redhat.jdg</groupId>
            <artifactId>hacep-core-camel</artifactId>
            <version>${hacep.version}</version>
        </dependency>

        <!-- HACEP Integration Dependencies -→
        <dependency>
            <groupId>com.redhat.refarch.hacep</groupId>
            <artifactId>integration-model</artifactId>
            <version>${project.version}</version>
        </dependency>
        <dependency>
            <groupId>com.redhat.refarch.hacep</groupId>
            <artifactId>integration-rules</artifactId>
            <version>${project.version}</version>
        </dependency>

        <!-- Logging Dependencies -→
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
```

```
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>${slf4j.version}</version>
        </dependency>
        <dependency>
            <groupId>log4j</groupId>
            <artifactId>log4j</artifactId>
            <version>${log4j.version}</version>
        </dependency>

        <!--Test Dependencies -→
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-all</artifactId>
            <version>${mockito.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

<dependencies>

    <!-- CDI Welding Dependencies -→
    <dependency>
        <groupId>javax.enterprise</groupId>
        <artifactId>cdi-api</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>javax.inject</groupId>
        <artifactId>javax.inject</artifactId>
        <scope>provided</scope>
    </dependency>
```

```
    <!-- JavaEE API Dependencies -→
    <dependency>
        <groupId>org.jboss.spec.javax.jms</groupId>
        <artifactId>jboss-jms-api_1.1_spec</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.transaction</groupId>
        <artifactId>jboss-transaction-api_1.1_spec</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.ejb</groupId>
        <artifactId>jboss-ejb-api_3.1_spec</artifactId>
        <scope>provided</scope>
    </dependency>
    <dependency>
        <groupId>org.jboss.spec.javax.ws.rs</groupId>
        <artifactId>jboss-jaxrs-api_1.1_spec</artifactId>
        <scope>provided</scope>
    </dependency>
</dependencies>
```

Lastly, define the *build* section responsible for configuring the maven plugins used:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>${maven.compiler.plugin.version}</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <maxmem>256M</maxmem>
                <showDeprecation>true</showDeprecation>
            </configuration>
        </plugin>
```

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-resources-plugin</artifactId>
        <version>${maven.resources.plugin.version}</version>
      </plugin>
    </plugins>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>${maven.exec.plugin.version}</version>
        </plugin>
        <plugin>
          <groupId>org.kie</groupId>
          <artifactId>kie-maven-plugin</artifactId>
          <extensions>true</extensions>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-war-plugin</artifactId>
          <version>${maven.war.plugin.version}</version>
          <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <version>${maven.surefire.plugin.version}</version>
          <configuration>
            <argLine>-Djava.net.preferIPv4Stack=true</argLine>
          </configuration>
        </plugin>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-dependency-plugin</artifactId>
```

```
                <version>${maven.dependency.plugin.version}</version>
                <executions>
                    <execution>
                        <id>copy-dependencies</id>
                        <phase>package</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${project.build.directory}/lib</outputDirectory>
                            <overWriteReleases>false</overWriteReleases>
                            <overWriteSnapshots>false</overWriteSnapshots>
                            <overWriteIfNewer>true</overWriteIfNewer>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>${maven.jar.plugin.version}</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-resources-plugin</artifactId>
                <version>${maven.resources.plugin.version}</version>
                <configuration>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </pluginManagement>

</build>
```

This concludes the defining and configuring of the parent project itself. At this time, stub out the 4 child modules which we'll be using inside the IDE. Rather than going through each inherent pom file of the child modules, please refer to the accompanying attachment *hacep-integration* folder and child module subfolders to locate and copy the contents of each pom file respectively.

### 5.5.2. The integration-model Module

This module, responsible for housing the shared POJO's and other Java objects necessary within both

the rules and main integration modules, consists of a skeleton *beans.xml* file (necessary for the triggering of CDI across the application) and only three other classes. Both the *integration-model* and *integration-app* modules will need a copy of this *beans.xml* file placed in the *src/main/resources/META-INF* directory, using the content below:

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
       java.sun.com/xml/ns/javaee/beans_1_0.xsd" >
</beans>
```

The other three classes within the integration-model module will fall into two packages:

- com.redhat.refarch.hacep.model
- com.redhat.refarch.hacep.model.channels

The model package will contain two classes, Purchase and PurchaseKey, which will serve as the Event POJO fed to the HACEP nodes via the Event Channel and the Key which will be used in classifying the *grouping* of the events into relevant rules sessions and node ownerships. Define Purchase object as an object extending the *Fact* interface from within the HACEP core code as follows:

```
import it.redhat.hacep.model.Fact;
import it.redhat.hacep.model.Key;

import java.time.Instant;
import java.util.Date;
import java.util.Objects;

public class Purchase implements Fact {

    private static final long serialVersionUID = 7517352753296362943L;

    protected long id;

    protected Long customerId;

    protected Date timestamp;

    protected Double amount;

    public Purchase(long id, Long customerId, Date timestamp, Double amount) {
        this.id = id;
        this.customerId = customerId;
        this.timestamp = timestamp;
        this.amount = amount;
    }

    @Override
    public Instant getInstant() {
        return timestamp.toInstant();
    }

    @Override
    public Key extractKey() {
        return new PurchaseKey(String.valueOf(id), String.valueOf(customerId));
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
```

```
    }

    public Long getCustomerId() {
        return customerId;
    }

    public void setCustomerId(Long customerId) {
        this.customerId = customerId;
    }

    public Date getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(Date timestamp) {
        this.timestamp = timestamp;
    }

    public Double getAmount() {
        return amount;
    }

    public void setAmount(Double amount) {
        this.amount = amount;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Purchase)) return false;
        Purchase purchase = (Purchase) o;
        return id == purchase.id &&
                Objects.equals(customerId, purchase.customerId) &&
                Objects.equals(timestamp, purchase.timestamp) &&
                Objects.equals(amount, purchase.amount);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, customerId, timestamp, amount);
    }

    @Override
    public String toString() {
        return "Purchase{" +
                "id=" + id +
                ", customerId=" + customerId +
                ", timestamp=" + timestamp +
```

```
                  ", amount=" + amount +
                  '}';
    }
}
```

Now define the PurchaseKey as seen below:

```
import it.redhat.hacep.model.Key;

import java.util.Objects;

public class PurchaseKey extends Key<String> {

    private String id;

    public PurchaseKey(String id, String customer) {
        super(customer);
        this.id = id;
    }

    public String getId() {
        return id;
    }

    public String toString() {
        return id + "::" + getGroup();
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof PurchaseKey)) return false;
        if (!super.equals(o)) return false;
        PurchaseKey purchaseKey = (PurchaseKey) o;
        return Objects.equals(id, purchaseKey.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(super.hashCode(), id);
    }
}
```

These two POJO's make up the only shared model needed for our simple integration. Lastly, we'll create a class called *SysoutChannel* in the channels package mentioned prior, which will serve as a pipeline for output from within rules sessions, which can be altered to conform to various logging needs. For our purposes, information sent to the sysout channel will simply be dumped to the console.

```
import org.kie.api.runtime.Channel;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SysoutChannel implements Channel {

    public static final String CHANNEL_ID = "sysout";
    private static final Logger LOGGER = LoggerFactory.getLogger(SysoutChannel.class);

    @Override
    public void send(Object object) {
        if (LOGGER.isInfoEnabled()) {
            LOGGER.info("=============================>" + object);
        }
    }
}
```

These three classes plus the beans.xml and pom files which have been created above complete the integration-model module.

### 5.5.3. The integration-rules Module

This module, like the integration-model module, is fairly simplistic and straight-forward. As with any BRMS integration, we'll define a *kmodule.xml* in the META-INF directory which defines the rules and sessions to be used. We'll also create two *DRL* rules files as well as one Java class containing a custom implementation of the rules accumulate function. Start by defining the two rules files in the *src/main/resources/rules* directory as follows:

- *purchase-audit.drl*: a simplistic base rule file which handles all the necessary function imports and rules event declarations.

```
package it.redhat.hacep.playground.rules.reward.catalog;

import com.redhat.refarch.hacep.model.Purchase;

import accumulate
com.redhat.refarch.hacep.rules.functions.ConsecutiveDaysAccumulateFunction
consecutivedays;

declare Purchase
    @role( event )
    @timestamp( timestamp.getTime() )
    @expires( 60d )
end
```

- *purchase-audit-rule1.drl*: secondary rule file defining two rules, one which simply echoes whenever a Purchase fact enters the session, and the other which will accumulate multiple purchases made by a customer so that we can reflect how many purchases by each repeat customer have been made over a 30-day period and the sum total of customer's purchases during the timeframe.

```
package it.redhat.hacep.playground.rules.reward.catalog;

import com.redhat.refarch.hacep.model.Purchase

rule "purchase seen"
when
    $purchase : Purchase ( )
then
    channels["sysout"].send("Purchase rec'd for customer " + $purchase.getCustomerId() +
" for amt " + $purchase.getAmount());
end

rule "report multiple purchases by a customer over 30-day window"
when
    $purchase : Purchase ( $customerId : customerId ) over window:length(1)
    $purchaseCount : Number( intValue > 1 )
                     from accumulate ($iter : Purchase ( $purchase.customerId ==
customerId ) over window:time( 30d ),
                     count( $iter ) )
    $purchaseSum : Number( )
                     from accumulate ( Purchase ( $purchase.customerId == customerId,
$purchaseAmount : amount ) over window:time( 30d ),
                     sum( $purchaseAmount ) )
then
    channels["sysout"].send("repeat customer " + $purchase.getCustomerId() + " has made "
+ $purchaseCount + " purchases over last 30 days for a sum of " + $purchaseSum);
end
```

Lastly, define the custom accumulate implementation in *src/main/java* under the *com.redhat.refarch.hacep.rules.functions* package as follows:

```
import org.kie.api.runtime.rule.AccumulateFunction;

import java.io.*;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.TimeUnit;

public class ConsecutiveDaysAccumulateFunction implements AccumulateFunction {

    @Override
    public void accumulate(Serializable context, Object value) {
        ConsecutiveDaysBuckets buckets = (ConsecutiveDaysBuckets) context;
        int days = getDays(value);
        if (buckets.buckets.get(days) == null) {
            buckets.buckets.put(days, new Integer(1));
        } else {
            buckets.buckets.put(days, buckets.buckets.get(days) + 1);
        }
    }

    @Override
    public Serializable createContext() {
        return new ConsecutiveDaysBuckets();
    }

    @Override
    public Object getResult(Serializable context) throws Exception {
        ConsecutiveDaysBuckets buckets = (ConsecutiveDaysBuckets) context;
        return buckets.buckets.size();
    }

    @Override
    public void init(Serializable context) throws Exception {
        ConsecutiveDaysBuckets buckets = (ConsecutiveDaysBuckets) context;
        buckets.buckets = new HashMap<>();
    }

    @Override
    public void reverse(Serializable context, Object value) throws Exception {
        ConsecutiveDaysBuckets buckets = (ConsecutiveDaysBuckets) context;
        int days = getDays(value);
        if (buckets.buckets.get(days) == null) {
```

```
            //ignore, shouldn't happen
        } else if (buckets.buckets.get(days) == 1) {
            buckets.buckets.remove(days);
        } else {
            buckets.buckets.put(days, buckets.buckets.get(days) - 1);
        }
    }

    @Override
    public boolean supportsReverse() {
        return true;
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        // TODO Auto-generated method stub

    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        // TODO Auto-generated method stub

    }

    @Override
    public Class<?> getResultType() {
        return Number.class;
    }

    public static class ConsecutiveDaysBuckets implements Externalizable {

        public Map<Integer, Integer> buckets;

        @Override
        public void writeExternal(ObjectOutput out) throws IOException {
            out.writeInt(buckets.size());
            for (int key : buckets.keySet()) {
                out.writeInt(key);
                out.writeInt(buckets.get(key));
            }
        }

        @Override
        public void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException {
            buckets = new HashMap<>();
            int size = in.readInt();
```

```
            for (int i = 0; i < size; i++) {
                buckets.put(in.readInt(), in.readInt());
            }
        }
    }

    private int getDays(Object value) {
        BigDecimal data = new BigDecimal(((Number) value).longValue());
        int days = data.divide(new BigDecimal(TimeUnit.DAYS.toMillis(1)), 2,
RoundingMode.HALF_UP).intValue();
        return days;
    }
}
```

These 4 files described above make up the entirety of the integration-rules module. Feel free to cross-reference the source code within the accompanying attachment for both structure and content accuracy as required.

### 5.5.4. The purchase-publisher Module

Next, define the classes within the helper module which will assist in create and submitting events to the HACEP-integrated application to simulate input from real use case scenarios. In a production system, such a tool wouldn't be used outside of perhaps testing purposes, but it fulfills the need for event creation that will allow observation and adminstrating of the HACEP framework in real-time. Add the following two classes to the purchase-publisher module under *src/main/java* in the *com.redhat.refarch.hacep.publisher* package:

- *PurchaseProducer*: class responsible for generating customer purchase events for submittal to the Event Channel broker.

```
import com.redhat.refarch.hacep.model.Purchase;

import javax.jms.*;
import java.math.BigDecimal;
import java.math.RoundingMode;
import java.util.Date;
import java.util.concurrent.ThreadLocalRandom;

public class PurchaseProducer {

    private final Connection connection;
    private final Session session;

    private String queueName;
    private long customerId;
    ConnectionFactory connectionFactory;
```

```
    private static final Double PURCHASE_MIN = 10.0;
    private static final Double PURCHASE_MAX = 500.0;


    public PurchaseProducer(ConnectionFactory connectionFactory, String queueName, long
customerId) {

        this.customerId = customerId;
        this.connectionFactory = connectionFactory;
        this.queueName = queueName;
        try {
            connection = connectionFactory.createConnection();
            session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }

    public void produce(Integer id, Long timestamp) {

        try {

            Queue destination = session.createQueue(queueName);
            MessageProducer producer = session.createProducer(destination);
            producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);

            Purchase fact = new Purchase(id, customerId, new Date(timestamp),
                    round(ThreadLocalRandom.current().nextDouble(PURCHASE_MIN,
PURCHASE_MAX)));

            System.out.println("sending purchase for customer " + customerId + " for
amount of " + fact.getAmount());

            ObjectMessage message = session.createObjectMessage(fact);
            message.setStringProperty("JMSXGroupID", String.format("P%05d", customerId));
            message.setIntProperty("JMSXGroupSeq", id);

            producer.send(message);

        } catch (Exception e) {
            System.out.println("Caught: " + e);
        }
    }

    public static double round(double value) {
        BigDecimal bd = new BigDecimal(value);
        bd = bd.setScale(2, RoundingMode.HALF_UP);
```

```
        return bd.doubleValue();
    }
}
```

- *Runner*: executable class responsible for establishing a connection to the Event Channel broker and orchestrating parallel thread submittals of Purchase content via PurchaseProducer. Note that there are some configurable options within the runner that should be modified to match your needs and environment.

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.pool.PooledConnectionFactory;

import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.TimeUnit;

public class Runner {

    private ScheduledExecutorService scheduler = null;

    private final String BROKER_URL =
"failover:(tcp://amq1:61616,tcp://amq2:61616,tcp://amq3:61616)";
    private final String QUEUE_NAME = "facts";
    private final String BROKER_USERNAME = "admin";
    private final String BROKER_PASSWORD = "admin";

    private final Integer CONCURRENT_PURCHASERS = 5;
    private final Integer DURATION = 15;
    private final Integer EVENT_DELAY = 15;
    private final Integer EVENT_INTERVAL = 5;
    private final Integer MAX_ACTIVE_SESSIONS = 250;
    private final Integer POOL_SIZE = 8;


    public Runner() {
        scheduler = Executors.newScheduledThreadPool(CONCURRENT_PURCHASERS);
    }

    public static void main(String[] args) throws Exception {
        new Runner().produce();
    }

    private void produce() throws InterruptedException {

        ActiveMQConnectionFactory amqConnectionFactory;
```

```
        amqConnectionFactory = new ActiveMQConnectionFactory(BROKER_URL);
        amqConnectionFactory.setUserName(BROKER_USERNAME);
        amqConnectionFactory.setPassword(BROKER_PASSWORD);

        PooledConnectionFactory connectionFactory = new
PooledConnectionFactory(amqConnectionFactory);
        connectionFactory.setMaxConnections(POOL_SIZE);
        connectionFactory.setMaximumActiveSessionPerConnection(MAX_ACTIVE_SESSIONS);

        for (int i = 0; i < CONCURRENT_PURCHASERS; i++) {

            int delay = (int) (Math.random() * EVENT_DELAY);

            final ScheduledFuture<?> playerHandle = scheduler
                    .scheduleAtFixedRate(
                            new PurchaseRunner(
                                    new PurchaseProducer(connectionFactory, QUEUE_NAME,
i)),
                                        delay,
                                        EVENT_INTERVAL,
                                        TimeUnit.SECONDS);

            scheduler.schedule(() -> playerHandle.cancel(true), DURATION,
TimeUnit.MINUTES);
        }
    }

    private class PurchaseRunner implements Runnable {

        private final PurchaseProducer purchaseProducer;
        private int id = 10000;

        public PurchaseRunner(PurchaseProducer purchaseProducer) {
            this.purchaseProducer = purchaseProducer;
        }

        @Override
        public void run() {
            purchaseProducer.produce(id++, System.currentTimeMillis());
        }
    }
}
```

### 5.5.5. The integration-app Module

Lastly, define the various components that make up the main module of the integration. This module will house several HACEP-inherent configurations, the HACEP integration itself, as well as a RESTful interface for monitoring of the application in action.

#### Resources

This main module will contain a a *log4j.properties* file underneath the *src/main/resources* directory and a *jboss-deployment-structure.xml* file underneath the *src/main/resources/WEB-INF* directory. These two files are fairly self explanatory, so reference the accompanying attachment source code in order to copy the content needed for them. A copy of the *beans.xml* file should already be present within *META-INF* from previous steps.

#### HACEP Configuration Files

The HACEP core code defines two interfaces which must be implemented in order to configure the framework. Within *src/main/java* add a *com.redhat.refarch.hacep.configuration* package with the following two classes:

- *DroolsConfigurationImpl*: responsible for the definition of rule package and session names, as well as channels utilized within the rules.

```
import com.redhat.refarch.hacep.model.channels.SysoutChannel;
import it.redhat.hacep.configuration.DroolsConfiguration;
import it.redhat.hacep.drools.channels.NullChannel;
import org.kie.api.KieBase;
import org.kie.api.KieServices;
import org.kie.api.runtime.Channel;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.enterprise.context.ApplicationScoped;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

@ApplicationScoped
public class DroolsConfigurationImpl implements DroolsConfiguration {

    private static final Logger logger =
LoggerFactory.getLogger(DroolsConfigurationImpl.class);

    private final Map<String, Channel> channels = new ConcurrentHashMap<>();
    private final Map<String, Channel> replayChannels = new ConcurrentHashMap<>();
    private final KieContainer kieContainer;
```

```
    private final KieBase kieBase;
    private static final String KSESSION_RULES = "hacep-sessions";
    private static final String KBASE_NAME = "hacep-rules";

    public DroolsConfigurationImpl() {
        KieServices kieServices = KieServices.Factory.get();
        kieContainer = kieServices.getKieClasspathContainer();
        kieBase = kieContainer.getKieBase(KBASE_NAME);
        channels.put(SysoutChannel.CHANNEL_ID, new SysoutChannel());
        replayChannels.put(SysoutChannel.CHANNEL_ID, new NullChannel());
        if (logger.isInfoEnabled()) {
            logger.info("[Kie Container] successfully initialized.");
        }
    }

    @Override
    public KieSession getKieSession() {
        return kieContainer.newKieSession(KSESSION_RULES);
    }

    @Override
    public KieBase getKieBase() {
        return kieBase;
    }

    @Override
    public Map<String, Channel> getChannels() {
        return channels;
    }

    @Override
    public Map<String, Channel> getReplayChannels() {
        return replayChannels;
    }

    @Override
    public int getMaxBufferSize() {
        try {
            return Integer.valueOf(System.getProperty("grid.buffer", "1000"));
        } catch (IllegalArgumentException e) {
            return 1000;
        }
    }
}
```

- *JmsConfigurationImpl*: responsible for referencing the Event Channel broker's JNDI connection and configuring the number of concurrent consumers to be used per HACEP node for taking input from the broker

```
import it.redhat.hacep.configuration.JmsConfiguration;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import javax.annotation.Resource;
import javax.enterprise.context.ApplicationScoped;
import javax.jms.ConnectionFactory;

@ApplicationScoped
public class JmsConfigurationImpl implements JmsConfiguration {
    private static final Logger LOGGER =
LoggerFactory.getLogger(JmsConfigurationImpl.class);

    @Resource(lookup = "java:/HacepConnectionFactory")
    private ConnectionFactory connectionFactory;

    @Override
    public ConnectionFactory getConnectionFactory() {
        if (LOGGER.isDebugEnabled()) {
            LOGGER.debug(String.format("Provide connection factory [%s]",
connectionFactory));
        }
        return connectionFactory;
    }

    @Override
    public String getQueueName() {
        try {
            return System.getProperty("queue.name", "facts");
        } catch (IllegalArgumentException e) {
            return "facts";
        }
    }

    @Override
    public int getMaxConsumers() {
        try {
            return Integer.valueOf(System.getProperty("queue.consumers", "5"));
        } catch (IllegalArgumentException e) {
            return 5;
        }
    }
}
```

### HACEP Instantiation & RESTful Interface

Within the *com.redhat.refarch.hacep* package, add the following two classes:

- *Application*: solely responsible for referencing and, thereby, instantiating the HACEP framework core code

```
import it.redhat.hacep.configuration.HACEPApplication;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.Singleton;
import javax.ejb.Startup;
import javax.inject.Inject;

@Startup
@Singleton
public class Application {

    @Inject
    private HACEPApplication hacepApplication;

    @PostConstruct
    public void start() {
        hacepApplication.start();
    }

    @PreDestroy
    public void stop() {
        hacepApplication.stop();
    }
}
```

- *ApplicationRestService*: defines a simplistic RESTful interface which would normally be a point of further expansion in production applications for interacting with the HACEP application. In this instance, two commands are exposed, the *info* command, which will provide basic information about the state of the node, the cluster, and the caches within the system, and *help* which provides a list of accepted commands via the *[IP_ADDR]:8080/execute/[command]* URL format.

```
import com.redhat.refarch.hacep.rest.RestInterface;
import com.redhat.refarch.hacep.rest.commands.InterfaceRequest;

import javax.inject.Inject;
import javax.ws.rs.*;
import javax.ws.rs.core.Application;
import javax.ws.rs.core.Response;
```

```
import java.util.Arrays;
import java.util.Collections;
import java.util.Optional;

@ApplicationPath("/")
@Path("/")
public class ApplicationRestService extends Application {

    @Inject
    private RestInterface restInterface;

    @GET
    @Path("/execute/{command}")
    @Produces("application/json")
    public Response executeCommand(@PathParam("command") String commandName,
@QueryParam("params") String params) {
        try {
            Optional<InterfaceRequest> command = restInterface.findByName(commandName);
            if (command.isPresent()) {
                if (params != null) {
                    command.get().execute(restInterface,
Arrays.asList(params.split(",")).iterator());
                } else {
                    command.get().execute(restInterface, Collections.emptyIterator());
                }
            } else {
                restInterface.printUsage();
            }
            return Response.ok(restInterface.getContent()).build();
        } finally {
            restInterface.clear();
        }
    }

    @GET
    @Path("/help")
    @Produces("application/json")
    public Response help() {
        restInterface.printUsage();
        return Response.ok(restInterface.toString()).build();
    }

    @GET
    @Path("/help/{command}")
    @Produces("application/json")
    public Response helpOnCommand(@PathParam("command") String commandName) {
        Optional<InterfaceRequest> command = restInterface.findByName(commandName);
        if (command.isPresent()) {
```

```
            command.get().usage(restInterface);
            return Response.ok(restInterface.toString()).build();
        }
        return help();
    }
}
```

### RESTful Interface Request Definitions

Within the *integration-app* module, the RESTful interface elicits a flexible & extendable set of commands for fetching and displaying information about the state of the application and its resources. These request commands follow a common interface so that they can be injected as one list and easily iterated from within the ApplicationRestService RESTful endpoints. The interface is as follows:

```
import com.redhat.refarch.hacep.rest.RestInterface;

import java.util.Iterator;

public interface InterfaceRequest {

    String command();

    boolean execute(RestInterface console, Iterator<String> args) throws
IllegalArgumentException;

    void usage(RestInterface console);
}
```

An example of one such interface implementation is the _ InfoInterfaceRequest_:

```
import com.redhat.refarch.hacep.dto.NodeType;
import com.redhat.refarch.hacep.dto.SessionDataObjectInformation;
import com.redhat.refarch.hacep.rest.RestInterface;
import com.redhat.refarch.hacep.util.JDGUtility;
import it.redhat.hacep.configuration.HACEPApplication;
import it.redhat.hacep.configuration.annotations.HACEPSessionCache;
import it.redhat.hacep.model.Key;
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.remoting.transport.Address;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import javax.inject.Inject;
import java.util.*;
```

```
public class InfoInterfaceRequest implements InterfaceRequest {

    private static final String COMMAND_NAME = "info";

    private final HACEPApplication application;

    @Inject
    private JDGUtility jdgUtility;

    @Inject
    @HACEPSessionCache
    private Cache<Key, Object> sessionCache;

    @Inject
    public InfoInterfaceRequest(HACEPApplication application) {
        this.application = application;
    }

    @Override
    public String command() {
        return COMMAND_NAME;
    }

    @Override
    public boolean execute(RestInterface console, Iterator<String> args) throws
IllegalArgumentException {
        try {
            DefaultCacheManager cacheManager = application.getCacheManager();
            console.println(generalInfo());
            for (String cacheName : cacheManager.getCacheNames()) {
                console.println("\n");
                console.println(buildInfo(cacheManager.getCache(cacheName)));
            }

            console.println("\n\nSession cache objects:");
            Map<Address, List<SessionDataObjectInformation>> sessions = new HashMap<>();
            for (Map.Entry<Key, List<Address>> entry :
jdgUtility.getKeysAddresses(sessionCache).entrySet()) {
                List<Address> addresses = entry.getValue() != null ? entry.getValue() :
Collections.emptyList();
                for (int i = 0; i < addresses.size(); i++) {
                    boolean isPrimary = (i == 0);
                    sessions.compute(addresses.get(i), (a, l) -> {
                        if (l == null) {
                            l = new ArrayList<>();
                        }
                        SessionDataObjectInformation object = new
```

```
SessionDataObjectInformation(entry.getKey().toString(), isPrimary ? NodeType.PRIMARY :
NodeType.REPLICA);
                        l.add(object);
                        return l;
                    });
                }
            }

            console.print(sessions);

        } catch (NoSuchElementException e) {
            console.println(generalInfo());
        }
        return true;

    }

    private String generalInfo() {
        DefaultCacheManager cacheManager = application.getCacheManager();
        StringBuilder info = new StringBuilder();
        info.append("Cache Manager Status:
").append(cacheManager.getStatus()).append("\n");
        info.append("Cache Manager Address:
").append(cacheManager.getAddress()).append("\n");
        info.append("Coordinator address:
").append(cacheManager.getCoordinator()).append("\n");
        info.append("Is Coordinator:
").append(cacheManager.isCoordinator()).append("\n");
        info.append("Cluster Name: ").append(cacheManager.getClusterName()).append("\n");
        info.append("Cluster Size: ").append(cacheManager.getClusterSize()).append("\n");
        info.append("Member list: ").append(cacheManager.getMembers()).append("\n");
        info.append("Caches: ").append(cacheManager.getCacheNames()).append("\n");
        return info.toString();
    }

    private String buildInfo(Cache<Key, Object> cache) {

        StringBuilder info = new StringBuilder();

        info.append("Cache: ").append(cache).append("\n");
        info.append("Cache Mode:
").append(cache.getCacheConfiguration().clustering().cacheModeString()).append("\n");
        info.append("Cache Size: ").append(cache.size()).append("\n");
        info.append("Cache Status: ").append(cache.getStatus()).append("\n");
        info.append("Number of Owners:
").append(cache.getAdvancedCache().getDistributionManager().getConsistentHash().getNumOwn
ers()).append("\n");
        info.append("Number of Segments:
```

```
").append(cache.getAdvancedCache().getDistributionManager().getConsistentHash().getNumSeg
ments()).append("\n");

        return info.toString();
    }

    @Override
    public void usage(RestInterface console) {
        console.println(COMMAND_NAME);
        console.println("\t\tGeneral information on caches.");
    }
}
```

For further examples of potential commands to add to the RESTful interface, reference and copy the *HelpInterfaceRequest* class from the *hacep-integration* source code within the accompanying attachment and also review the *it.redhat.hacep.playground.console.commands* package within the *hacep-eap-playground* module of the HACEP source code.

### Utility Classes for Diagnostics of HACEP Caches and Sessions

The *com.redhat.refarch.hacep.dto* package will contain a simple Enum and object that will assist in carrying information about the HACEP caches and sessions forward when the *info* command is called upon. Their content is as follows:

- *NodeType*

```
public enum NodeType {
    PRIMARY, REPLICA;
}
```

- *SessionDataObjectInformation*

```
public class SessionDataObjectInformation {

    private String name;
    private NodeType nodeType;

    public SessionDataObjectInformation() {
    }

    public SessionDataObjectInformation(String name, NodeType nodeType) {
        this.name = name;
        this.nodeType = nodeType;
    }

    public String getName() {
        return name;
    }

    public NodeType getNodeType() {
        return nodeType;
    }
}
```

## 5.6. Packaging & Deploying the Integration Application

Once code-complete, packaging and deployment of the application follows. Start by navigating to the application's root *hacep-integration* directory and issuing the *mvn clean package* command to kick off building and compilation of the war & jar files of the child modules where applicable.

Once building is complete and successful, the next step is deploying the *integration-app*.war file by copying it from the *integration-app/target* directory to the *standalone/deployments* directory of each HACEP EAP node you wish to run the application on. Upon deployment, server logs should reflect various steps of CDI wiring, connecting to the A-MQ Event Channel, Infinispan JGroups cluster coordination, and initialization of the BRMS rules sessions.

## 5.7. Executing the Purchase Publisher for Real-Time Observation

Once the publisher Runner class has been customized, maven packaging has been completed, and the integration application has been deployed to the various HACEP nodes, we can now use the *purchase-publisher* module to generate and feed events to the Event Channel. Within the project source folder, navigate to the *hacep-integration/purchase-publisher/target* directory, which should now contain an executable *jar* file, and issue the following command to commence event generation:

```
java -Dorg.apache.activemq.SERIALIZABLE_PACKAGES="*" -cp purchase-publisher-1.0-
SNAPSHOT.jar com.redhat.refarch.hacep.publisher.Runner
```

## 5.8. Summation

The modules making up the *hacep-integration* project described above make up the entirety of the simple HACEP integration application, discounting the *purchase-publisher* module, which as previously mentioned serves a purpose for proof-of-concept, but doesn't necessarily pertain to production functionality. The model and rules modules both serve as utility modules supplying the rules definitions and object models required, while the *integration-app* module acts as the HACEP integration and configuration point and exposes some simple monitoring functionality via a RESTful interface, demonstrating how interaction and monitoring of such an application could be achieved in HACEP-integrated systems.

# 6. Conclusion

This reference architecture reviews the architectural design and features of the `HACEP`, or Highly-Available and Scalable Complex Event Processing, framework. The technologies used within the framework are then discussed in detail as to best describe how the framework functions.

Following, examples provided alongside the HACEP source code which exemplify usage of HACEP directly via command-line and then via REST API within an EAP-hosted application are also previewed and discussed. Creation of an example enterprise application integration where HACEP is paired with JBoss A-MQ as an event channel is also described step-by-step.

By walking through the architectural design and integration of the HACEP framework, various features, points of consideration, and best practices are outlined and presented in circumstances similar to those which will be encountered when leveraging the framework in every-day situations and production systems.

# Appendix A: Revision History

| Revision | Release Date | Author(s) |
| --- | --- | --- |
| 1.1 | August 2016 | Jeremy Ary |
| 1.0 | August 2016 | Jeremy Ary, Ugo Landini, and Fabio Marinelli |

# Appendix B: Contributors

We would like to thank the following individuals for their time and patience as we collaborated on this process. This document would not have been possible without their many contributions.

| Contributor | Title | Contribution |
| --- | --- | --- |
| Ugo Landini | Senior Solution Architect | HACEP Framework Author & Technical Content Review |
| Fabio Marinelli | Senior Architect | HACEP Framework Author & Technical Content Review |
| Babak Mozaffari | Manager, Software Engineering & Consulting Engineer | Technical Content Review |