Red Hat Reference Architecture Series

Real-Time Access to Big Data with Red Hat's JBoss Data Grid

Babak Mozaffari
Consulting Software Engineer
Systems Engineering

Version 1.0
August 2014

redhat.

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

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 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.

Staying In Touch

Join us on some of the popular social media sites where we keep our audience informed on new reference architectures as well as offer related information on things we find interesting.

Like us on Facebook:

https://www.facebook.com/rhrefarch

Follow us on Twitter:

https://twitter.com/RedHatRefArch

Plus us on Google+:

https://plus.google.com/114152126783830728030/

Table of Contents

1 Executive Summary

With the advent of modern big data technology, businesses require faster access to larger amounts of data to retain their competitive edge. Fast changing business environments and processes demand agile development practices that can keep lower overhead to survive through challenging times and keep up with growth to take full advantage of market opportunities.

Red Hat JBoss Data Grid is a distributed in-memory data grid, providing the ability to:[1]

- Handle unprecedented transaction volumes.
- Meet high uptime requirements.
- Deploy into hybrid cloud environments.
- Quickly access accurate, real-time information.
- Streamline interaction with complex and rigid data tiers.

This reference architecture reviews **`Red Hat JBoss Data Grid (JDG) 6.3`**, configures a cluster of JDG servers operating in remote client-server mode, and walks through the design, implementation and deployment of a replicated and distributed cache, a data model and applications that access and update the cache.

To demonstrate the power of JBoss Data Grid in the context of a realistic use case, a fictitious business called *Gas Shopper* is considered and a prototype is built to address the business requirements while taking advantage of a JDG cluster.

Gas Shopper uses a combination of business listings and crowd-sourced data to provide its customers with a list of nearby gas stations with up to date gas prices and sorted by distance. As a startup testing the market, *Gas Shopper* is starting out locally and gradually rolling out to the rest of the nation. If successful, there are plans to expand globally and even offer similar data and prices for products other than gasoline.

As a startup, the operating business does not have the budget to acquire and build a large datacenter. At the same time, they cannot afford to be unprepared. JBoss Data Grid allows the application to start out with a small cluster of 3 nodes, optionally hosted on a public cloud that can reduce the cost of hardware when there is little load. Scaling up is almost instantaneous as nodes can be added to the cluster and the deployment can move on-premise or elsewhere as needed. Once the application is rolled out globally, JDG allows data centers to replicate data across different geographic locations. While JBoss Data Grid can persist data in a database or the file system, it is also a great fit for transient data such as gas prices that does not need to be stored long term. Cache expiration policy can be used to provide memory management and automatically remove old data.

1 http://www.redhat.com/products/jbossenterprisemiddleware/data-grid/

2 Red Hat JBoss Data Grid 6.3

2.1 Overview

Red Hat JBoss Data Grid (JDG) 6.3 is a scalable, highly available data store and data grid platform, implemented as an open source, java-based product. Some of the capabilities provided by **JDG** include:[2]

- Schema-less key-value store – JBoss Data Grid is a NoSQL database that provides the flexibility to store different objects without a fixed data model.
- Grid-based data storage – JBoss Data Grid is designed to easily replicate data across multiple nodes.
- Elastic scaling – Adding and removing nodes is simple and non-disruptive.
- Multiple access protocols – It is easy to access the data grid using Hot Rod, REST, Memcached, or simple map-like API.

JBoss Data Grid may be used as a distributed cache, where higher performance and faster access to the data is desired and the data is stored in a database, disk-based NoSQL store or another mechanism. Data consistency may depend on cache coherency and the distributed capabilities of **JDG** can provide such coherency.

Another common use for **JBoss Data Grid** is as a NoSQL data store. While primarily an in-memory store, **JDG** also supports persisting data through cache stores. Various cache store implementations are provided with the product, making it possible to create a data store backed by the file system directly, a JDBC-based database or even another remote **JDG** cluster that receives calls through the **Hot Rod** protocol.

JBoss Data Grid can be used in both *Remote Client Server Mode* as well as *Library Mode*.[3]

Remote Client-Server mode provides a managed, distributed, and clusterable data grid server. It is packaged as a server platform and used through Hot Rod, Memcached or REST protocols.

In library mode, the data grid is accessed through local Java calls and used as a Java library by the business application.

2 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Getting_Started_Guide/chap-Red_Hat_JBoss_Data_Grid.html

3 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Getting_Started_Guide/sect-Red_Hat_JBoss_Data_Grid_Usage_Modes.html

2.2 Usage Mode

Red Hat JBoss Data Grid may be used in either remote client-server mode or library mode. Business requirements and architectural preferences help determine the suitable usage mode for an application. At a high level, one approach treats the grid or cache as a black box through a remote protocol, while the other uses it as an embedded library.

All data grid operations in remote client-server mode are non-transactional. As a result, a number of features cannot be performed when running JBoss Data Grid in this mode. However there are also a number of benefits to running in the remote client-server mode when those features are not required. The client-server mode is client language agnostic, provided there is a client library for your chosen protocol. This provides easier scaling of the data grid, as well as easier upgrades of the data grid without impact on client applications.

On the other hand, library mode allows building and deploying a custom runtime environment. The library mode hosts a single data grid node in the applications process, with remote access to nodes hosted in other JVMs. Features that require transactions, listeners and notifications are only available in library mode.

Tested containers for Red Hat JBoss Data Grid 6 library mode includes Red Hat JBoss Web Server 2 and JBoss Enterprise Application Platform 6. JBoss Data Grid can also be run as a standalone application in Java SE, avoiding a container altogether.

For further information about the usage modes for the JBoss Data Grid server, refer to the official Red Hat documentation.[4]

2.2.1 Remote Client-Server Mode

To use JBoss Data Grid in remote client-server mode, first download the server archive file from Red Hat's Customer Support Portal.[5] For version 6.3.0, the download is called:

- Red Hat JBoss Data Grid Server 6.3.0

After the download, simply unzip the archive file to set up the data grid server. The archive file for version 6.3.0 is *jboss-datagrid-6.3.0-server.zip*.

The data grid server is started by running the *standalone* script for a single instance and the *clustered* script for two or more data grid instances. Both scripts are located in the *bin/* directory. In remote client-server mode, JBoss Data Grid provides the following APIs:

2.2.1.1 The REST Interface

The data grid server can be accessed through a simple REST API over HTTP. The main benefit of this interface is its simplicity and portability, as REST over HTTP is a common and standard method of communication that is supported in most environments with little to no additional library requirements.

4 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Getting_Started_Guide/sect-Red_Hat_JBoss_Data_Grid_Usage_Modes.html

5 https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=data.grid&productChanged=yes

Adding data to the grid through REST is accomplished through a simple HTTP PUT or HTTP POST call. To provide a value under a key called myKey in a cache called myCache, provide the value as the request body in a call to the following URL:

http://myserver:port/rest/myCache/myKey

In a PUT call, any preexisting value for myKey would be silently replaced and updated with the provided request body. With POST, the call is only successful if the entry does not exist. An existing myKey would result in a 409 HTTP error code, indicating a conflict.

Sending a GET call to this same URL return the stored cache value corresponding to the key. The returned Content-Type header indicates the type of the value data. Other information returned includes a Last-Modified header.

The response of a HEAD call to the same URL returns the same headers as the GET call, but without any response body.

To delete a key-value pair, send a DELETE call to the same URL:

http://myserver:port/rest/myCache/myKey

Excluding the key would remove the entire cache. Including a performAsync header with the value of true results in asynchronous behavior where the call is returned immediately and the deletion happens in the background.

For further information about the REST interface to the JBoss Data Grid server, refer to the official Red Hat documentation.[6]

2.2.1.2 The Memcached Interface

Memcached is an in-memory caching system used to improve response and operation times for database-driven websites. The Memcached caching system defines a text based protocol called the Memcached protocol. The Memcached protocol uses in-memory objects or (as a last resort) passes to a persistent store such as a special memcached database.

Red Hat JBoss Data Grid offers a server that uses the Memcached protocol, removing the necessity to use Memcached separately with JBoss Data Grid. Additionally, due to JBoss Data Grid's clustering features, its data failover capabilities surpass those provided by Memcached.

For further information about the Memcached interface to the JBoss Data Grid server, refer to the official Red Hat documentation.[7]

2.2.1.3 The Hot Rod Interface

Hot Rod is a binary TCP client-server protocol used in Red Hat JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached.

By using a binary protocol instead of text-based, JBoss Data Grid accelerates server and client communication.

6 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-The_REST_Interface.html

7 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-The_Memcached_Interface.html

Hot Rod clients receive the cluster topology in the form of updates from the cluster and use this information to fail over as appropriate, when a server fails.

In partitioned or distributed JBoss Data Grid server clusters, Hot Rod clients retrieve the same consistent hash algorithm as the server and this allows the client to directly communicate with a cluster node containing the desired cache data. This is another way that Hot Rod increases efficiency.

Using the Cache Manager API with the Hot Rod protocol also enables the use of the Asynchronous API. This allows the client to avoid blocking calls and retrieve responses at a later time.

2.2.2 Library Mode

In the library mode, JBoss Data Grid is used as a simple Java library that is embedded in the client application. For applications that uses **Maven**, the easiest starting point is to include a dependency on the data grid module. Set the version as appropriate:

```
<dependency>
          <groupId>org.infinispan</groupId>
          <artifactId>infinispan-core</artifactId>
          <version>${infinispan.version}</version>
</dependency>
```

The cache itself may be configured either declaratively or programmatically.

For example, the following code programmatically configures a cache cluster that uses synchronous distribution with two owners for the data:

```
new ConfigurationBuilder()
   .clustering()
      .cacheMode(CacheMode.DIST_SYNC)
      .hash().numOwners(2)
   .build()
```

Alternatively, the same configuration may be expressed in XML:

```
<default>
   <clustering mode="distribution">
      <sync/>
      <hash numOwners="2"/>
   </clustering>
</default>
```

Access to the cache typically occurs through the *Cache Manager* interface and would therefore be largely identical regardless of whether an *Embedded Cache Manager* is used in library mode or a Remote Cache Manager to access a grid or cache located elsewhere, as would typically be the case when using the remote client-server mode. For further information about Cache Managers, refer to the official Red Hat documentation.[8]

8 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-Cache_Managers.html

2.3 Memory Management

2.3.1 Eviction

Eviction is the process of removing entries from the cache or grid, there by clearing some of the memory. In JBoss Data Grid, eviction is an intelligent process that can be used to keep the most relevant data in memory while moving less immediate items to disk through a configured cache store, to prevent permanent data loss.

Red Hat JBoss Data Grid executes eviction proactively, based on the cache configuration and not reactively due to low JVM memory.

Various eviction parameters are configured, starting with the eviction strategy. Depending on the strategy, other parameters help further define the eviction behavior. For example a strategy of *LRU* (least recently used) means the least recently used cache entries are evicted and *max entries* determines the number of the recent cache entries that are preserved where the rest are evicted.

For further information about eviction, refer to the official Red Hat documentation.[9]

2.3.2 Passivation

When eviction is used with a persistent cache, the persistent store (if a cache store is configured) always retains all the data while the memory grows and shrinks as data is loaded and evicted. With frequently referenced data, this model can result in expensive repetitive writes to the cache store.

Passivation, used in conjunction with eviction, prevents the maintenance of a duplicate persisted copy of the data and instead uses the persistent store as an overflow tank, where evicted data is persisted when not in use and removed from persistence when loaded. This avoids constant write operations when data is available in cache and frequently used.

For further information about passivation, refer to the official Red Hat documentation.[10]

2.4 Expiration

While eviction and passivation are attempts at memory management, expiration deals with the management of data itself. An expired cache entry (or even the entire cache itself) is not moved from dynamic memory to a persistent store, but rather removed globally and permanently so that it will no longer exist in the memory, cluster or cache store.

Expiration can be specified per cache entry or per cache itself, where the finer grained setting overrides the coarser configuration. Expiration may be configured based on lifespan or idle time, respectively counting the time since the cache entry was created or the time elapsed since it was last used.

9 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-Set_Up_Eviction.html

10 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/part-Set_Up_Passivation.html

Any cache entry that lacks expiration configuration on itself or its parent cache is called an immortal data. Alternatively, mortal data is automatically removed by JBoss Data Grid after a finite period of time.

For further information about expiration, refer to the official Red Hat documentation.[11]

2.5 Cache Modes

2.5.1 Overview

Red Hat JBoss Data Grid may be used in either local mode or clustered mode. In clustered mode, more than one node participate and communicate with one another to replicate, invalidate or distribute the data, depending on the configuration.

For further information about cache modes, refer to the official Red Hat documentation.[12]

2.5.2 Local Mode

In local mode, JBoss Data Grid is simply running on one virtual machine. In such setups, various cluster capabilities of JBoss Data Grid go unused and at first glance the cache may look like a simple Java Map, however a JDG cache provides various features that justify its use, even with a single node:

- Write-through and write-behind caching to persist data
- Entry eviction to prevent the Java Virtual Machine (JVM) running out of memory
- Support for entries that expire after a defined period

2.5.3 Replication Mode

Replication mode is the commonly understood setup for a clustered cache, where a number of nodes participate in a cluster and replicate the data throughout the nodes, resulting in each entry being available locally on any node.

JBoss Data Grid can make use of UDP multicast to improve performance in a large cluster, however using replication is generally not recommended for a very large number of nodes and performs best with a maximum of ten nodes.

Replication mode can be synchronous or asynchronous depending on the problem being addressed.[13]

- Synchronous replication blocks a thread or caller (for example on a put() operation) until the modifications are replicated across all nodes in the cluster. By waiting for acknowledgments, synchronous replication ensures that all replications are

11 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-Set_Up_Expiration.html

12 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-Synchronous_and_Asynchronous_Replication.html

13 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-Synchronous_and_Asynchronous_Replication.html

successfully applied before the operation is concluded.

- Asynchronous replication operates significantly faster than synchronous replication because it does not need to wait for responses from nodes. Asynchronous replication performs the replication in the background and the call returns immediately. Errors that occur during asynchronous replication are written to a log. As a result, a transaction can be successfully completed despite the fact that replication of the transaction may not have succeeded on all the cache instances in the cluster.

Replicating data across four nodes, with A to H representing various parts of data, would therefore look as follows:

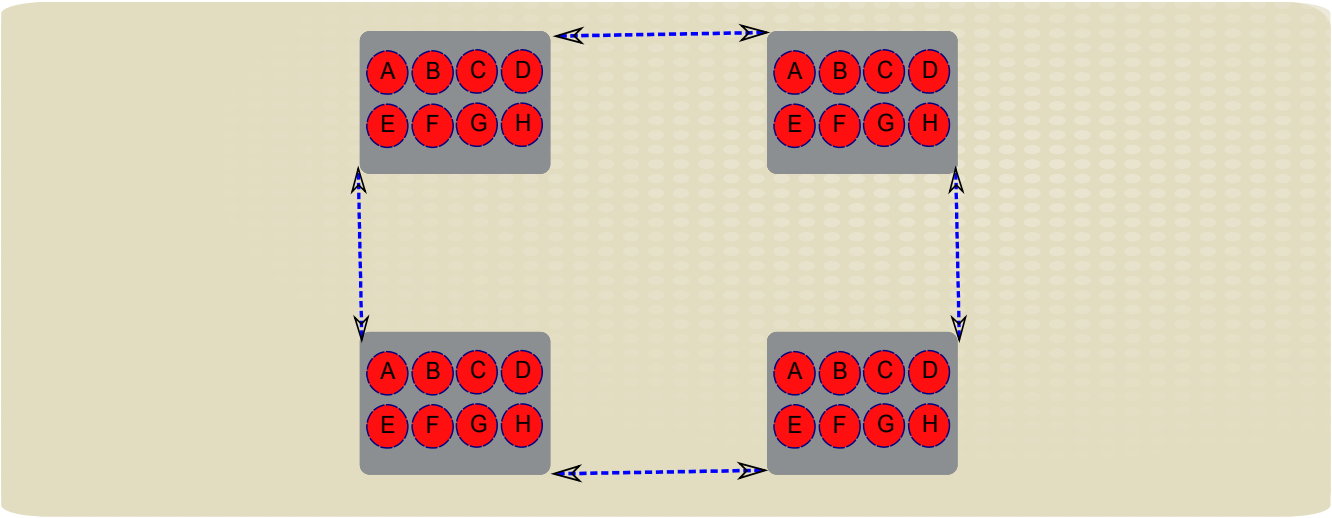

Figure 2.5.3-1: Replication across multiple nodes

2.5.4 Distribution Mode

In distribution mode, two or more nodes work together to form a data grid. Instead of each node containing the exact same data, with entries replicated uniformly, the data is distributed among the nodes to increase scalability. The result is remarkably efficient horizontal scaling as adding nodes increases storage capacity linearly while maintaining a constant response time.

JBoss Data Grid uses a consistent hash algorithm to determine the node or nodes that should hold any given cache entry. When a client calls a node, requesting a cache entry that is not locally available, the hash algorithm is used by the local node to locate the data and a synchronous call retrieves the data and seamlessly returns it to the calling client. With certain client protocols, namely the Hot Rod protocol, an intelligent client is able to use the same consistent hash algorithm to directly locate the data and contact the correct node in order to avoid extra calls and increase performance.

In distribution mode, JBoss Data Grid can be configured so that each cache entry is stored in a given number of nodes. This helps provide redundancy and fault tolerance without hindering scalability.

Distributing data across four nodes with two owners would therefore look as follows:

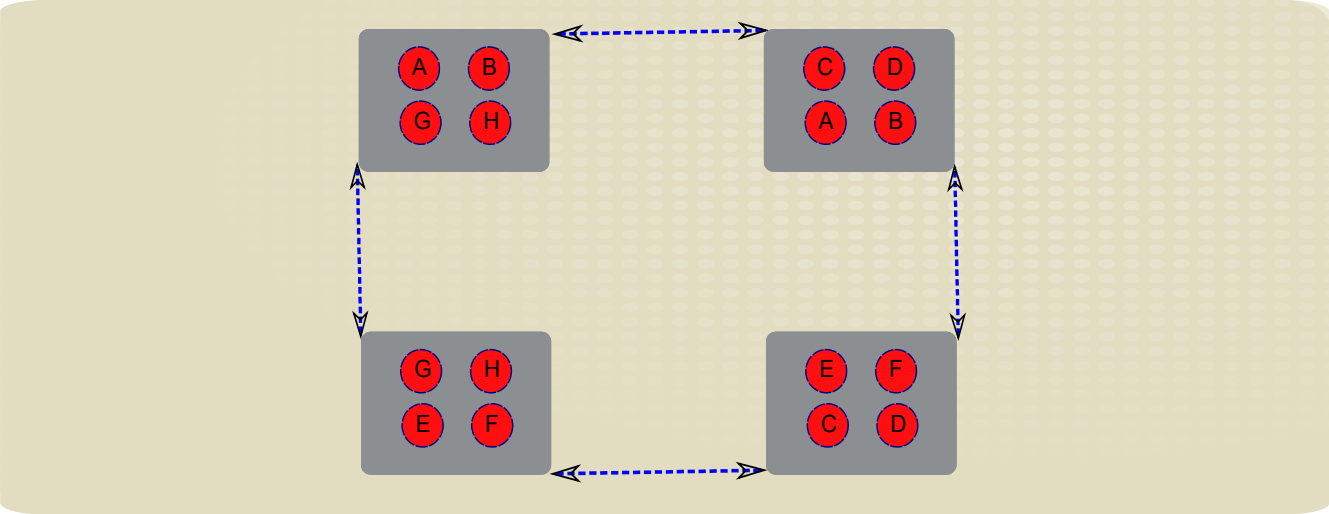

Figure 2.5.4-1:

JBoss Data Grid provides a set of advanced features for controlling the behavior of data distribution.

Server Hinting ensures that backed up copies of data are not stored on the same physical server, rack, or data center as the original.[14]

The key affinity service provides more control to developers by allowing the placement of a value in a certain node in a distributed cluster.[15]

Grouping allows separate cache entries to be logically grouped, so that they are always stored on the same nodes of the cluster.[16]

2.5.5 Invalidation Mode

Invalidation mode assumes a more permanent store for the data, where JBoss Data Grid is only acting as a cache, typically to increase performance.

With invalidation mode, changes in data result in a signal across the cluster to remove obsolete entries and avoid stale data from being returned. This means that upon the next request for a data item that has been invalidated in a node, a cache miss occurs, which typically results in reading the permanent data store and storing the updated value in the cache.

14 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-High_Availability_Using_Server_Hinting.html
15 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-Key_Affinity_Service.html
16 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Developer_Guide/chap-The_Grouping_API.html

2.6 Locking

To prevent concurrencies issues and dirty reads, JBoss Data Grid provides cluster-wide locking mechanisms.

JBoss Data Grid supports both optimistic and pessimistic locking. With optimistic locking, the extra overhead of locking cache entries across a cluster is avoided and it is assumed that concurrent modification attempts will not occur. This is useful for scenarios where contention is low and the cost of pessimistic locking is not justified. With optimistic locking, a *writeSkew* check occurs and results in a rollback and error if the data was found to have been concurrently modified.[17]

Pessimistic locking is also known as eager locking. This mechanism notifies all cluster members that a node is or will be attempting to modify one or more cache entries and prevents any other transaction from modifying those entries. Pessimistic locking can be explicit where entries are locked through the API, for example:

```
cache.lock(K)
```

Pessimistic locking may also happen implicitly as a result of a cache modification. Writing a cache entry causes a pessimistic lock, if it has been configured but not explicitly applied already:

```
cache.put(K,V)
```

2.7 Cache Loaders

The cache loader provides Red Hat JBoss Data Grid's connection to a persistent data store. The cache loader retrieves data from a data store when the required data is not present in the cache.

Cache Loaders are primarily intended for use cases where JDG is used as a classic cache, front-ending a persistent data store, typically for performance reasons.

JBoss Data Grid provides a number of cache loaders that are ready to use as well as the ability to implement your own cache loader by implementing the provided interface.

Cache loaders may be configured declaratively or programmatically. Several configuration options are available, including the option to preload data, which eagerly fetches the cache data and makes it available immediately after startup. Other configurations determine the synchronous or asynchronous behavior of a cache loader, how it behaves in a cluster along with other cache loader instances and how it reacts to data modification.

For further information about cache loaders, refer to the official Red Hat documentation.[18]

17 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-Locking_Types.html#Optimistic_Locking

18 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-Cache_Stores.html#Cache_Loaders_and_Cache_Writers

2.8 Cache Stores

The cache store connects Red Hat JBoss Data Grid to a persistent data store. The cache store is used to:

- fetch data from the data store when a copy is not in the cache.
- push modifications made to the data in cache back to the data store.

Caches that share the same cache manager can have different cache store configurations, as cache stores are associated with individual caches.

Cache stores are implementations of the *CacheWriter* interface, which extends the *CacheLoader* interface and adds writing capability to it.

The *SingleFileCacheStore* is a simple, file system based implementation of the cache store that is provided along with JBoss Data Grid. This cache store stores all key/value pairs and their corresponding metadata information in a single file.

Due to the limitations of the file system, the use of the *SingleFileCacheStore* is not supported in all production environments. For example, certain shared file systems (such as NFS and Windows shares) lack proper file locking, resulting in data corruption. Furthermore, file systems are not inherently transactional, resulting in file writing failures during the commit phase if the cache is used in a transactional context.

Another cache store implementation provided with JBoss Data Grid is the *RemoteCacheStore*. the *RemoteCacheStore* is essentially a delegate to a remote JBoss Data Grid cluster. It uses the fast and efficient Hot Rod protocol to make remote calls to a different Data Grid, loading data from it when required and storing its data as needed.

Various configuration parameters are available for the remote cache store to control the TCP behavior, cluster location and other such parameters.

Custom cache stores may also be implemented and configured, but are not supported by Red Hat.

Finally, a high performance cache store provided with JBoss Data Grid is the *LevelDB* cache store. *LevelDB* is a key-value storage engine that provides an ordered mapping from string keys to string values. The *LevelDB* Cache Store uses two filesystem directories. Each directory is configured for a *LevelDB* database. One directory stores the non-expired data and the second directory stores the expired keys before a purge.

For information on setup and configuration of the *LevelDB* cache store, refer to the official Red Hat documentation.[19]

19 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-LevelDB_Cache_Store.html

2.9 Cache Writing

When a cache store is configured, inserting, modifying or removing cache entries may propagate to the backing persistent data store. This process is known as cache writing.

JBoss Data Grid supports two cache writing paradigms:

- Write-Through (Synchronous)
- Write-Behind (Asynchronous)

With synchronous write-through cache writing, an attempt by the client to update the cache, typically through a simply cache.put() call, results in instant persistence of the data. The API call block and waits until the data is written through to the persistent data store and thereby ensures data consistency, as any issues in writing the data to the store would have been already detected and proper remedies can therefore be taken at the time.

Alternatively, asynchronous write-behind cache writing avoids the cost of writing to disk and immediately returns the API call to modify the cache entry. Instead, cache store updates are queued and carried out by a thread different.

While write-behind cache writing is generally reliable, it does result in a brief period of time when the cache store contains stale data compared to the cache.

For further information on cache writing and configuration details, refer to the official Red Hat documentation.[20]

2.10 Monitoring

Monitoring of various metrics is supported, including the hit ratio of the cache, the average read or write time, evictions, removals and so on. Similar useful metrics are also returned for replication, invalidation, activation and passivation occurring for the cache.

For further information on cache monitoring, configuring and using JBoss Operations Network or JMX Mbeans, and monitoring cache managers and instances, refer to the official Red Hat documentation.[21]

2.10.1 Remote Client-Server Mode

In Red Hat JBoss Data Grid's Remote Client-Server mode, the JBoss Operations Network plug-in is used to:

- initiate and perform installation and configuration operations.
- monitor resources and their metrics.

JBoss Operations Network (JON) provides high level metrics on the cache manager including the Cache Container Status and the Cluster Name, as well as metrics on the caches

20 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/chap-Cache_Writing_Modes.html

21 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/part-Monitor_Caches_and_Cache_Managers.html

themselves such as cache status, average read time, hit ratio, read/write ratio and many others.

For further details on the JON monitoring capabilities for JDG in remote client-server mode, refer to the official Red Hat documentation.[22]

Red Hat documentation for JBoss Data Grid also includes the necessary instructions for installing JON plug-ins.[23]

2.10.2 Library Mode

Java Management Extension (JMX) is a Java based technology that provides tools to manage and monitor applications, devices, system objects, and service oriented networks. Each of these objects is managed, and monitored by *MBeans*.

JMX is the de-facto standard for middleware management and administration. As a result, JMX is used in Red Hat JBoss Data Grid to expose management and statistical information.

Management in Red Hat JBoss Data Grid instances aims to expose as much relevant statistical information as possible. This information allows administrators to view the state of each instance. While a single installation can comprise of tens or hundreds of such instances, it is essential to expose and present the statistical information for each of them in a clear and concise manner.

In JBoss Data Grid, JMX is used in conjunction with **`JBoss Operations Network (JON)`** to expose this information and present it in an orderly and relevant manner to the administrator.

JMX statistics can be enabled at two levels:

- At the cache level, where management information is generated by individual cache instances.
- At the CacheManager level, where the CacheManager is the entity that governs all cache instances created from it. As a result, the management information is generated for all these cache instances instead of individual caches.

Refer to official Red Hat documentation for using JON with JBoss Data Grid in library mode.[24]

2.11 Data Grid CLI

Red Hat JBoss Data Grid includes the Red Hat JBoss Data Grid Command Line Interface (CLI) that is used to inspect and modify data within caches and internal components (such as transactions, cross-datacenter replication sites, and rolling upgrades). The JBoss Data Grid CLI can also be used for more advanced operations such as transactions.

---

22 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-JBoss_Operations_Network_Remote-Client_Server_Plugin.html

23 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-JBoss_Operations_Network_for_Remote_Client-Server_Mode.html

24 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-JBoss_Operations_Network_for_Library_Mode.html

The CLI consists of a server-side module and a client command tool. In the remote client-server mode, simply starting the JBoss Data Grid Server also starts up the server side CLI module. To connect to the server and issue CLI commands, run the *bin/ispn-cli* script. The JMX address of the server can be provided to this script along with a -c flag.

In CLI mode, use the *cache* command followed by the name of a cache to specify it as the default, to which the subsequent CLI commands would apply.

Transactions can be demarcated in CLI with the *begin* and *end* commands. Cache entries can be retrieved by issuing a *get* command and added with a *put* command.

For example to create two cache entries with the keys of *a* and *b* in a transaction, and the same values of *a* and *b*:

```
begin
put a a
put b b
end
```

To see the result:

```
get a
get b
```

CLI commands can also be used for administrative use cases, to display the configuration of a selected cache or container, create a new cache based on the configuration of an existing cache, abort a batch or roll back a transaction.

For a complete list of supported CLI commands for the JBoss Data Grid, refer to the official Red Hat documentation.[25]

2.12 Rolling Upgrades

An important administrative command is the *upgrade* command, which implements the rolling upgrade procedure. In Red Hat JBoss Data Grid, rolling upgrades permit a cluster to be upgraded from one version to a new version without experiencing any downtime. This allows nodes to be upgraded without the need to restart the application or risk losing data.

Rolling upgrades can be performed in Remote Client-Server mode using Hot Rod and REST.

Refer to the developer guide for further details on rolling upgrades.[26]

25 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/part-Command_Line_Tools.html

26 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Developer_Guide/chap-Rolling_Upgrades.html

3 Reference Architecture Environment

3.1 Overview

This reference architecture sets up a JBoss Data Grid cluster with two nodes, used in remote client-server mode, to store and provide data through Hot Rod and other protocols.

The application tier may be hosted on **`JBoss Enterprise Application Server (EAP)`** but the generic setup and configuration of the application server is outside the scope of this reference architecture. For information on installing and configuring a JBoss EAP cluster, refer to the JBoss EAP 6 Clustering reference architecture. Clients with access to the Red Hat Customer Portal may download the reference architecture and attachments from https://access.redhat.com/site/articles/524633

A public version of this document, without the file attachment, is available from http://www.redhat.com/resourcelibrary/reference-architectures/jboss-eap-6-clustering

Standalone Java code is also provided and used to populate static data as well as sample cache entries.

Two separate caches are configured in the JDG cluster and while the initial and configured cluster size is two nodes in a single data center, expansion to a larger cluster and another geographically remote data center is foreseen and described.

The **`LevelDB`** file-based cache store is used to provide efficient and high performant persistence for both caches.

3.2 JBoss Data Grid Cluster

This reference architecture stands up and configures a cluster of two JDG nodes, acting in remote client-server mode. The JDG server uses lightweight components from JBoss EAP to run in server mode and provides remote access to the data grid using Hot Rod, Memcached or REST client APIs (over http). Remote Client-Server mode provides a managed, distributed, and clusterable data grid server.

The cluster is configured as two JDG nodes running on separate machines and using UDP to communicate with one another. Both caches are set up in compatibility mode to allow REST queries as well as the Hot Rod protocol used by the provided application.

3.2.1 Replicated Cache

Two separate caches are configured in this reference architecture. The first cache contains a much smaller number of cache entries and uses significantly less memory. The data that is stored in this cache is largely read-only and rarely changes. This cache is configured to be replicated since the cost of full replication is very low, even with a much larger cluster size.

3.2.2 Distributed Cache

The second cache used in this reference architecture stores crowd-sourced data. While starting small, the user base for the application is expected to grow and the cache capacity

would have to grow with it.

This cache is the primary driver of the requirement to use JDG and its exponential grow may lead to a need to significantly scale up the JDG cluster by adding a large number of nodes. Expansion to international markets may also justify the setup of JDG cluster nodes in separate data centers for each given geography.

3.3 Web Application

To demonstrate the use of JBoss Data Grid in a practical use case, this reference architecture designs, develops and ships a Web Application that acts as a client to both the replicated and distributed caches.

The Web Application may be deployed on a cluster or single node of EAP 6. For information on installing and configuring an EAP 6 cluster, refer to the corresponding reference architecture as mentioned in the Overview.

The Web Application bundles and uses Hot Rod client libraries to make efficient binary calls to the JDG server.

3.4 Standalone Client

Standalone Java code is developed to load the initial data into the replicated cache as well generate sample data and load it into the distributed cache.

The standalone Java code is provided as an executable JAR file. In addition to the required bytecode, the source Java classes are also included in the JAR archive.

4 Creating the Environment

4.1 Prerequisites

Prerequisites for creating this reference architecture include a supported Operating System and JDK. Refer to Red Hat documentation for supported environments.[27]

4.2 Downloads

The attachments to this document provide copies of the JDG server configuration file, the required module descriptors and Java classes and archive files. These files may be downloaded from:

https://access.redhat.com/node/1155333/40/0

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

Download JBoss Data Grid Server 6.3.0 from Red Hat's Customer Support Portal:[28]

- Red Hat JBoss Data Grid Server 6.3.0

The standalone Java application provided with this reference architecture can load gas station data from comma-separated files into the cache. This code has been tested with three gas station data files available for download at the following location: http://www.essostations.com/gps_garmin_down2.php

Once downloaded, extract the comma-separated values (CSV) files from the archive.

To read the CSV files, the standalone Java application provided with this reference architecture makes use of the **opencsv** Java library. Download the project binary files from the following location:

http://sourceforge.net/projects/opencsv/files/opencsv/2.3/

Once the archive file has been downloaded, extract *deploy/opencsv-2.3.jar* from the archive.

The standalone Java application also uses the **Gson** Java library to parse the JSON response returned by the REST API of JBoss Data Grid servers. Download the project binaries from the following location:

https://code.google.com/p/google-gson/downloads/detail?name=google-gson-2.2.4-release.zip

Once the archive file has been downloaded, extract *gson-2.2.4.jar* from the archive.

27 https://access.redhat.com/site/articles/115883
28 https://access.redhat.com/jbossnetwork/restricted/listSoftware.html?downloadType=distributions&product=data.grid&version=6.3.0

4.3 Installation

4.3.1 JBoss Data Grid Server

JBoss Data Grid Server is provided in archived zip file format. Copy this archive file to your preferred location and extract it:

```
# unzip jboss-datagrid-6.3.0-server.zip
```

No further installation task is necessary. A folder called jboss-datagrid-6.3.0-server is created in the specified location and serves as the JDG home. This reference architecture document assumes that the folder is created at */opt/jboss-datagrid-6.3.0-server*

4.3.2 JBoss Enterprise Application Platform

While this reference architecture does not directly require JBoss EAP, a sample web application is designed and developed to interact with the JDG caches and needs to be deployed to an application server. For further details, refer to the section on the Web Application.

4.4 Configuration

4.4.1 Operating System Environment

In the reference environment, a few ports are used for intra-node communication or to access the cache. This includes port 11222, used by the Hot Rod protocol and accessed from the Web Application or any other Hot Rod client. Port 8080 is also used when accessing the cache through REST / HTTP. Port 54200 may be used for cluster failure detection and ports 4447, 4712 and 4713 are intended for transactions and remoting services; they are opened as well even if not immediately used in the reference architecture. Multicast requires UDP communication to be permitted and the firewall is configured accordingly. This reference architecture uses **`IPTables`**, the default Red Hat Firewall, to block all network packets by default and only allow configured ports and addresses to communicate. Refer to the Red Hat documentation on **`IPTables`**[29] for further details and see the appendix on IPTables configuration for the firewall rules used for the JDG cluster in this reference environment.

This reference environment has been set up and tested with **`Security-Enhanced Linux (SELinux)`** enabled in *ENFORCING* mode. Once again, refer to the Red Hat documentation on SELinux for further details on using and configuring this feature.[30] For any other operating system, consult the respective documentation for security and firewall solutions to ensure that maximum security is maintained while the ports required by your application are opened.

Various other types of configuration may be required for UDP and TCP communication. For example, **`Linux`** operating systems typically have a low maximum socket buffer size configured, which is lower than the default cluster **`JGroups`** buffer size. It may be important to correct any such warnings observed in the EAP logs. For example, in this case for a **`Linux`** operating system, the maximum socket buffer size may be configured by editing the */etc/sysctl.conf* file. Further details are available on Red Hat's Customer Support Portal.[31]

```
# Allow a 25MB UDP receive buffer for JGroups
net.core.rmem_max = 26214400
# Allow a 1MB UDP send buffer for JGroups
net.core.wmem_max = 1048576
```

To have the new values take effect without requiring a server reboot, enter the following command:

```
# sysctl -p
```

29 https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security_Guide/sect-Security_Guide-IPTables.html
30 https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Security-Enhanced_Linux/
31 https://access.redhat.com/site/solutions/190643

4.4.2 JBoss Data Grid Server

This reference architecture runs JBoss Data Grid servers in clustered mode using the default script. As such, the server configuration file in effect is *standalone/configuration/clustered.xml*. Replace this file with the equivalent provided in the attached archive file under *jdg/standalone/configuration/*.

For JBoss Data Grid to store objects of custom types in a cache, you must make the corresponding Java classes available. These Java classes are packaged as a JAR file called *gasshopper.jar* and provided as a module named *com.redhat.refarch.jdg.gasshopper*. Copy the *com/redhat/refarch/jdg/gasshopper/main* directory structure provided in the attachment under *jdg/modules* to the *modules/* directory of your JDG servers.

Update the **`infinispan`** module in your JDG servers to declare a dependency on the custom types by overwriting the provided module descriptor for *org.infinispan.commons* in the JDG servers with the following file provided in the reference architecture attachments:

jdg/modules/system/layers/base/org/infinispan/commons/main/module.xml

Create an application user with the role REST on each JDG server where the cache needs to be accessed through the REST interface. Creating an application user requires running the *add-user* script in interactive mode. Follow this process on each node of the JDG cluster:

```
What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): b
```

- Enter *b* and press enter to add an application user

```
Enter the details of the new user to add.
Realm (ApplicationRealm) :
```

- Simply press enter to continue

```
Username : Babak
```

- Enter the username as *Babak* press enter

```
Password : password1!
```

- Enter "*password1!*" as the password, and press enter

```
Re-enter Password : password1!
```

- Enter "*password1!*" again to confirm, and press enter

```
What roles do you want this user to belong to? (Please enter a comma
separated list, or leave blank for none)[  ]: REST
```

- *Type the role as REST* and press enter

```
About to add user 'Babak' for realm 'ApplicationRealm'
Is this correct yes/no? yes
```

- Type *yes* and press enter to continue

```
...[confirmation of user being added to files]
Is this new user going to be used for one AS process to connect to another
AS process?
e.g. for a slave host controller connecting to the master or for a Remoting
connection for server to server EJB calls.
yes/no? no
```

- Type *no* and press enter to complete user setup

At this point, the JBoss Data Grid cluster may be started by simply running the provided clustered script. It is good practice to always explicitly bind a server to the desired interface. To run a JDG server on a machine with an IP address of 10.19.139.101, run:

```
# ./clustered.sh -b 10.19.139.101
```

4.4.3 JBoss Enterprise Application Server

The sample web application is provided as an archive file called GasShopperWeb.war. Deploy the application to the EAP server or servers by placing this file in the *standalone/deployments* directory or following an alternative deployment approach.

While starting the EAP 6 server, provide a Java system property to point to one of the JDG servers. For example, assuming that the EAP server runs on 10.19.139.100 and the JDG cluster spans 10.19.139.101, 10.19.139.102 and 10.19.139.103, the EAP server in standalone mode can be started as follows:

```
# ./standalone.sh -b 10.19.139.100 -Djdg.host=10.19.139.101
```

4.4.4 Standalone Client

The standalone client is provided as a runnable JAR file called *GasShopperClient.jar*. The JDG client libraries required to execute this JAR file are provided in a directory called *GasShopperClient_lib* that must be located alongside the Java archive.

The client may be used to populate the cache with gas station data found and downloaded from the internet. To load the Esso gas stations into the cache along with generated price history mockups, download the required third party files as explained in the Downloads section and copy the *gson-2.2.4.jar* and *opencsv-2.3.jar* files into the *GasShopperClient_lib* directory. Make the CSV files available on the file system so that they can be provided as command line arguments to the JAR file.

4.5 Execution

4.5.1 JBoss Data Grid Cluster

Start JBoss Data Grid on each node of the cluster by explicitly binding it to the desired interface on that node. For example, this reference environment runs a JDG cluster of three nodes on the IP addresses 10.19.139.101, 10.19.139.102 and 10.19.139.103.

Start the server on node1 by running:

```
# ./clustered.sh -b 10.19.139.101
```

Look at the standard output or server log to see the formed cluster composition. At this point, assuming that the node is called *eap-cluster-node1*, the last cluster view message looks as follows:

```
Received new cluster view: [eap-cluster-node1/clustered|0] (1) [eap-cluster-
node1/clustered]
```

Proceed by starting the server on nodes 2 and 3. As the first server to be started, node1 will now be acting as the coordinator. Look for the following messages on the output or log of the node1 server as nodes 2 and 3 are started up:

```
Received new cluster view: [eap-cluster-node1/clustered|1] (2) [eap-cluster-
node1/clustered, eap-cluster-node2/clustered]
```

```
Received new cluster view: [eap-cluster-node1/clustered|2] (3) [eap-cluster-
node1/clustered, eap-cluster-node2/clustered, eap-cluster-node3/clustered]
```

It is helpful to monitor the server logs for cluster view messages to detect any communication problems between the cluster nodes.

4.5.2 Sample Data Population

To populate sample data, follow the previous instructions and make sure the following items have been downloaded:

- CSV files containing gas station data
- GasShopperClient and its library folder, included in the attachments
- OpenCSV JAR file

Place *opencsv-2.3.jar* in the client library folder where other dependency jar libraries reside. Make sure that the CSV file are available from the location that the client code is executed.

The GasShopperClient load function uses the Hot Rod protocol to contact a JDG server on the associated port, set to 11222 by default. This means that the client can run from any machine that can access a JDG server but given the very large amount of initial data populated in the cache, it is strongly recommended that you would load the sample data from the same machine running the JDG host that you are targeting.

To load gas station data from all 3 CSV files that you have downloaded and create ten mockup price reports for each gas station, run the client as follows:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar load
./esso.csv,./exxon_usa.csv,./mobil_usa.csv
```

The “-D” is used to pass a system parameter to the JVM and in this case, *jdg.host* is set to the first node of the JDG cluster where this command is also executed from. The port is not provided and the default port of 11222 is assumed. The client JAR file is referenced and its libraries are assumed to be in the designated directory. The CSV files are all provided at once with a comma separator, although they can also be provided separately in different runs.

4.5.3 Web Application

As part of its setup, this reference environment install an EAP 6 server and deploys the attached web application on that server. The server is configured to point to node1 of the JDG cluster. It is normally best to avoid using a single point of failure in the JDG cluster but this setup suffices for demonstration purposes. It should also be noted that after initialization, the servlet in the web application maintains an active reference to both remote caches through the Hot Rod protocol, which includes a cluster-aware stub that is notified of any changes in the cluster topology. This is similar to a JNDI lookup for an Enterprise Java Bean, where it is still best to target multiple servers for the JNDI lookup but it is a one-time action and subsequent failure of that primary node has no impact on future calls.

To avoid a single point of failure on first contact, add multiple servers to the configuration builder:

```
builder.addServer().host( host ).port( Integer.valueOf( port ) );
```

Point your browser to the web application deployed on EAP 6:

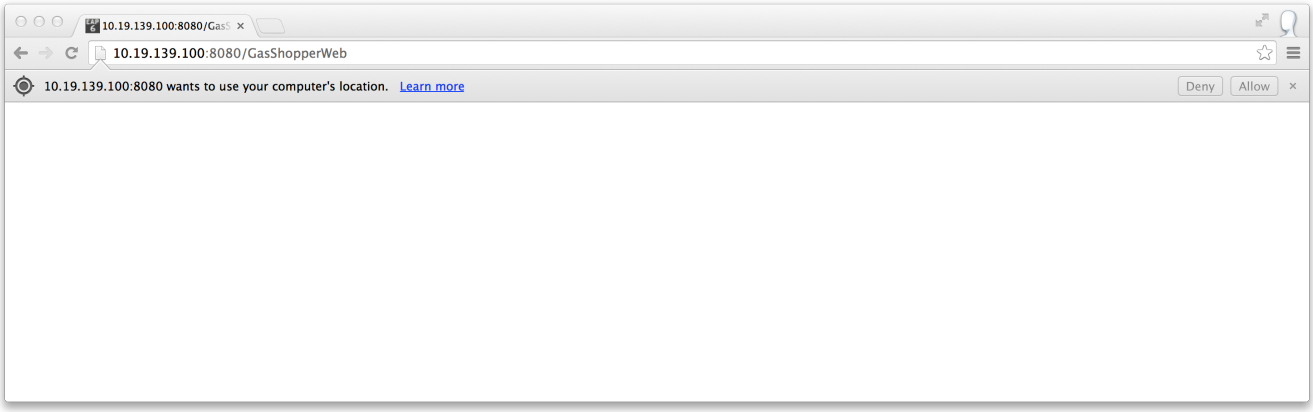

Figure 4.5.3-1

The first time the web application is reached, JavaScript code is loaded and executed to determine the user's location. Depending on your browser's security setting, you might be prompted to agree to share your location. Allow the application to retrieve your location, This results in a redirect where the browser's best assumption for your geographical coordinates are sent back to the web application servlet. The web application then contacts the cache and retrieves all the gas stations near you:

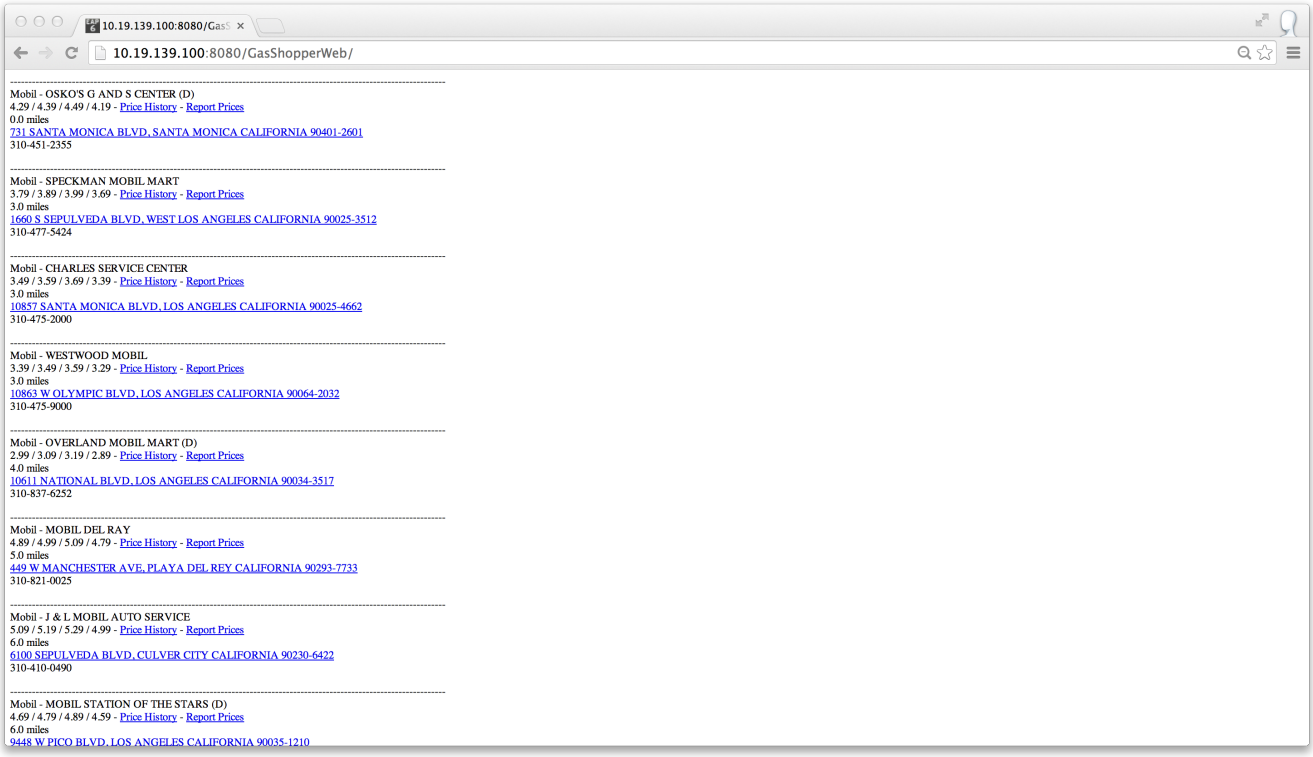

Figure 4.5.3-2

The number of gas stations retrieved will depend on your estimated location. The web application code finds all the grid cells within a 20 mile radius of your estimated location and displays all the gas stations in those cells, even if they are much farther away than 20 miles.

The gas stations are sorted by distance, with those on top being the nearest.

The first line printed for each gas station is the name, consisting of the brand and station name. The most recent price report for the gas station, normally presumed to be the current price, is printed on the second line, with the price of regular, midgrade, premium and diesel printed in the respective order. This is followed by two links, each opening a new browser window.

Clicking on *Price History* retrieves all available price reports for the gas station from the cache. The price reports are sorted from most recent to least recent and include the exact date and time of the report as well as the IP address of the user who has reported the price:

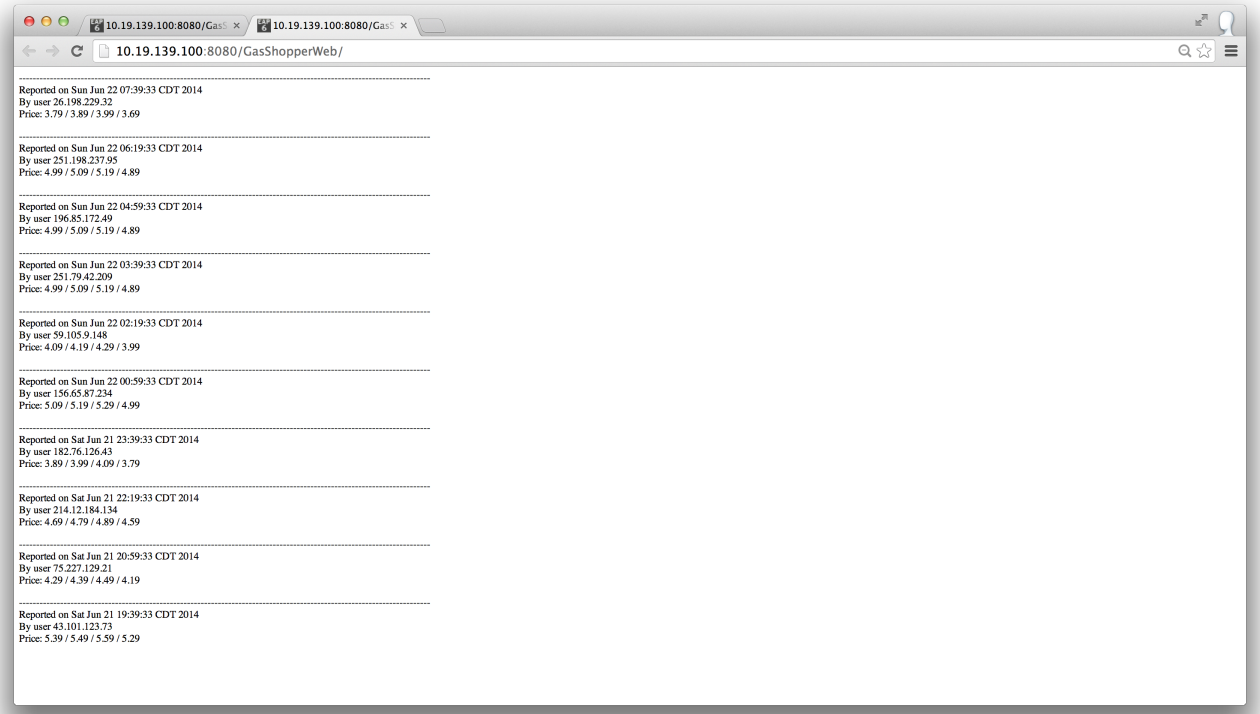

Figure 4.5.3-3

The second link allows you to report updated prices. A blank form collects prices for all 4 grades of fuel and submits them to the cache as the most recent price report for the selected gas station. The linked list of price reports is updated so that this report points to the previously most recent report. Your IP address will be recorded along with the prices.

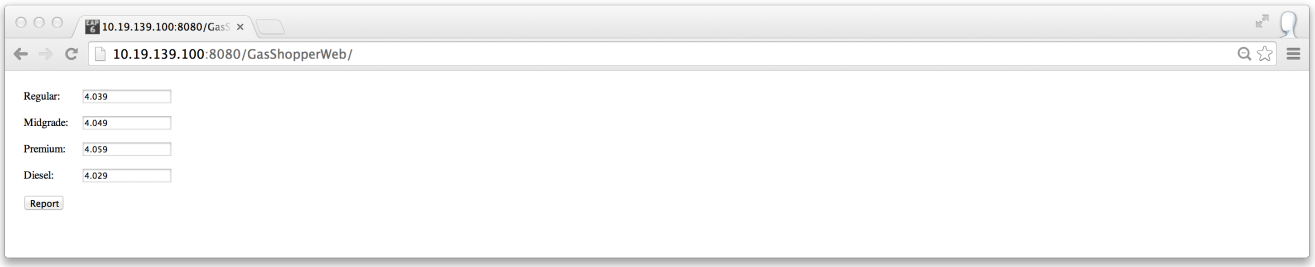

Figure 4.5.3-4

4.5.4 Miscellaneous

4.5.4.1 Overview

Other than loading gas station data and generating mockup price reports, the Standalone Client also demonstrates the use of REST over HTTP to retrieve data from the cache as well as the use of Hot Rod to get the size of cache held at the current node and to compare it with the total size of the cache stored on the grid.

For a full list of parameter options accepted by the client JAR file, execute it without any parameters:

```
# java -jar GasShopperClient.jar
```

The response:

```
Provide appropriate arguments as follows:

To load gas station date from files and generate mockup price reports:
     load file1.csv,file2.csv,file3.csv...

To query all gas stations within identified grid cell through REST:
     stations latitude longitude username password

To get price reports through REST, most recent priceReportId is always
stationId. Next id will be in the returned report:
     prices priceReportId username password

For information on the size of each cache:
     size
```

4.5.4.2 Retrieving Gas Stations through REST

To retrieve all the gas stations stored in the *stations* cache as one unit, provide the geographical coordinates of the center of the grid cell in question. Remember that both the latitude and longitude will always be whole numbers.

Along with coordinates, also provide the credentials of a user with the security role of REST, as required for REST over HTTP calls and explained in the section on configuring JBoss Data Grid Server:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar stations 34 -118
Babak password1!
```

The REST URL that is used for the HTTP request is provided, along with the response code:

```
----------------------------------------
Executing GET http://10.19.139.101:8080/rest/stations/(34.0,-118.0)
----------------------------------------
----------------------------------------
200 OK
----------------------------------------
```

The above coordinates cover a densely populated area of Los Angeles and therefore include 262 of the gas stations provided in the sample data download, as of the date of writing. To avoid printing too much information to the console, the client prints out the full details of a single gas station and only prints the stations IDs of the remaining stations:

```
REST response parsed to 262 gas stations, here is a random example:
GasStation [company=Mobil - LACIENEGA MOBIL, address=2305 S LACIENEGA BLVD,
LOS ANGELES CALIFORNIA 90034-1609, telephone=310-839-7361,
coordinates=Coordinates [latitude=34.03797, longitude=-118.37761],
stationId=9f2ceb0e-a4ed-4019-af4a-10760f11f9b5]


All station UUIDs:
*************************
9f2ceb0e-a4ed-4019-af4a-10760f11f9b5
e632f3de-8c5b-407e-8e63-0328c1108f62
fdeab7a8-e143-453b-a674-2eb58e100397
c6d480ec-ad0f-4ed1-a23c-8c1e2be6a51a
42365614-1ec3-447a-9caf-6d0c77a04d58
421edf4f-94a0-471a-a610-c61a0ef57bee
ead790f3-5185-4689-8dee-5e7dc2735fde
2fc7b76a-a432-40b3-8820-7107abba42b3
5b3002b4-329b-4644-ab16-8cc91cc75933
5941eade-cfaa-46a9-b95c-5e245139dee8
aee3c2f3-52d6-4fa4-afa5-cab5d67b1dd3
afab653f-95d6-4853-b2bb-e7a4419279e2
c78146d6-eb41-477b-8082-52da37fd7767
375353ed-c696-4d2c-a646-e242762fc261
...
...
```

Also note that the client code requests the response in JSON format and uses the *gson* library to parse it back to the provide object types before printing the information. In fact, demonstrating the JSON format and parsing the response is the main reason for including this capability in the client code. Otherwise, it is very easy to retrieve the response through a REST call using basic command line tools:

```
# wget --user=Babak --password=password1!
                    http://10.19.139.101:8080/rest/stations/\(34.0,-118.0\)
```

The following is printed in the console:

```
--2014-06-24 09:26:31--  http://10.19.139.101:8080/rest/stations/(34.0,-
118.0)
Connecting to 10.19.139.101:8080... connected.
HTTP request sent, awaiting response... 401 Unauthorized
Reusing existing connection to 10.19.139.101:8080.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/xml]
Saving to: “(34.0,-118.0)”

    [ <=>  ] 134,240     --.-K/s   in 0.09s
2014-06-24 09:26:31 (1.41 MB/s) - “(34.0,-118.0)” saved [134240]
```

The actual response is saved in a file under the name *(34.0,-118.0)*. The file contains all 262 gas station details in XML format:

```
<map>
  <entry>
    <uuid>9f2ceb0e-a4ed-4019-af4a-10760f11f9b5</uuid>
    <com.redhat.refarch.jdg.gasshopper.GasStation>
      <company>Mobil - LACIENEGA MOBIL</company>
      <address>2305 S LACIENEGA BLVD, LOS ANGELES CALIFORNIA 90034-
1609</address>
      <telephone>310-839-7361</telephone>
      <coordinates>
        <latitude>34.03797</latitude>
        <longitude>-118.37761</longitude>
      </coordinates>
      <stationId reference="../../uuid"/>
    </com.redhat.refarch.jdg.gasshopper.GasStation>
  </entry>
  <entry>
    <uuid>e632f3de-8c5b-407e-8e63-0328c1108f62</uuid>
    <com.redhat.refarch.jdg.gasshopper.GasStation>
      <company>Mobil - J &amp;amp; L MOBIL AUTO SERVICE</company>
      <address>6100 SEPULVEDA BLVD, CULVER CITY CALIFORNIA 90230-
6422</address>
      <telephone>310-410-0490</telephone>
      <coordinates>
        <latitude>33.98335</latitude>
        <longitude>-118.39432</longitude>
      </coordinates>
      <stationId reference="../../uuid"/>
    </com.redhat.refarch.jdg.gasshopper.GasStation>
  </entry>
  <entry>
    <uuid>fdeab7a8-e143-453b-a674-2eb58e100397</uuid>
    <com.redhat.refarch.jdg.gasshopper.GasStation>
      <company>Mobil - QUINTIN&amp;apos;S MOBIL (D)</company>
      <address>1004 N HACIENDA BLVD, LA PUENTE CALIFORNIA 91744</address>
      <telephone>626-333-4791</telephone>
      <coordinates>
        <latitude>34.03748</latitude>
        <longitude>-117.94963</longitude>
      </coordinates>
      <stationId reference="../../uuid"/>
    </com.redhat.refarch.jdg.gasshopper.GasStation>
  </entry>
  <entry>
    <uuid>c6d480ec-ad0f-4ed1-a23c-8c1e2be6a51a</uuid>
    <com.redhat.refarch.jdg.gasshopper.GasStation>
      <company>Mobil - SAM&amp;apos;S MOBIL</company>
      <address>1600 N EASTERN AVE, LOS ANGELES CALIFORNIA 90063-
1018</address>
...
...
```

4.5.4.3 Retrieving Price Reports through REST

To retrieve price reports for a given gas station, start requesting the latest price report by using the station's ID. The client code can use REST over HTTP, asking for the response in JSON format and parsing it to the provided custom *PriceReport* type:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar prices
                9f2ceb0e-a4ed-4019-af4a-10760f11f9b5 Babak password1!
```

Once again, the REST URL and the response code are printed:

```
----------------------------------------
Executing GET http://10.19.139.101:8080/rest/prices/9f2ceb0e-a4ed-4019-af4a-
10760f11f9b5
----------------------------------------
----------------------------------------
200 OK
----------------------------------------
```

The response itself is a single price report in JSON:

```
{"ipAddress":"82.150.191.36","date":1403607627569,"regular":4.99,"midgrade":
5.09,"premium":5.19,"diesel":4.89,"lastReportKey":"9f2ceb0e-a4ed-4019-af4a-
10760f11f9b5_1403602827569"}
```

The *gson* library is used to parse this object into a *PriceReport*, which is then printed using its *toString()* method:

```
REST response parsed to PriceReport object:
**************************
PriceReport [ipAddress=82.150.191.36, date=Tue Jun 24 06:00:27 CDT 2014,
regular=4.99, midgrade=5.09, premium=5.19, diesel=4.89,
 lastReportKey=9f2ceb0e-a4ed-4019-af4a-10760f11f9b5_1403602827569]
**************************
```

To make a simple *wget* call and get the response in XML:

```
# wget --user=Babak --password=password1!
http://10.19.139.101:8080/rest/prices/9f2ceb0e-a4ed-4019-af4a-10760f11f9b5
```

```
--2014-06-24 09:44:49--  http://10.19.139.101:8080/rest/prices/9f2ceb0e-
a4ed-4019-af4a-10760f11f9b5
Connecting to 10.19.139.101:8080... connected.
HTTP request sent, awaiting response... 401 Unauthorized
Reusing existing connection to 10.19.139.101:8080.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [application/xml]
Saving to: “9f2ceb0e-a4ed-4019-af4a-10760f11f9b5.1”
    [ <=> ] 366         --.-K/s   in 0s
2014-06-24 09:44:49 (62.3 MB/s) - “9f2ceb0e-a4ed-4019-af4a-10760f11f9b5.1”
saved [366]
```

The actual response is saved in a file under the name *9f2ceb0e-a4ed-4019-af4a-10760f11f9b5*. The file contains the latest price report for the gas station, in XML format:

```
<com.redhat.refarch.jdg.gasshopper.PriceReport>
  <ipAddress>82.150.191.36</ipAddress>
  <date>2014-06-24 11:00:27.569 UTC</date>
  <regular>4.99</regular>
  <midgrade>5.09</midgrade>
  <premium>5.19</premium>
  <diesel>4.89</diesel>
  <lastReportKey>9f2ceb0e-a4ed-4019-af4a-10760f11f9b5_1403602827569
                                                    </lastReportKey>
</com.redhat.refarch.jdg.gasshopper.PriceReport>
```

Remember that price reports are saved as a linked list. The *lastReportKey* value above is the cache key to the report before last:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar prices
    9f2ceb0e-a4ed-4019-af4a-10760f11f9b5_1403602827569 Babak password1!
```

The parsed response:

```
REST response parsed to PriceReport object:
**************************
PriceReport [ipAddress=124.160.27.166, date=Tue Jun 24 04:40:27 CDT 2014,
regular=3.09, midgrade=3.19, premium=3.29, diesel=2.99,
lastReportKey=9f2ceb0e-a4ed-4019-af4a-10760f11f9b5_1403598027569]
**************************
```

This chain continues until either a null *lastReportKey* is encountered because you have reached the very first price report for the gas station, or there is no value for the cache key because it has expired and been evicted from the cache. When a price report is evicted from the cache, all the earlier reports would have naturally also expired so there is no concern with the linked list breaking.

4.5.4.4 Cache Size

Finally, passing the size argument to the client returns cache size information. The client code uses the Hot Rod protocol to get the size of the cache through two different approaches:

Calling *RemoteCache.size()* returns the size of the cache on the target node. For a replicated node or even a distributed node where the number of cluster members is not larger than the configured number of owners, the size of the cache on the target node is the total size of the cache. However in a distributed cache, the number of cluster members is usually larger than the configured number of owners and as a result, all cache entries do not reside on all nodes. The *prices* cache is one such case. The client code makes a second call as follows to obtain the total size of the cache: *RemoteCache.keySet().size()*. With this second call, first all the cache keys are requested and as would be expected of a data grid, the data is aggregated from the entire grid to return a result that is both complete and non-redundant. The number of keys is the actual size of the cache:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar size
```

The bold sections of the response have been formatted for clarity:

```
Jun 24, 2014 6:00:10 AM org.infinispan.client.hotrod.impl.protocol.Codec10
readNewTopologyAndHash
INFO: ISPN004006: /10.19.139.103:11222 sent new topology view (id=4)
containing 3 addresses: [/10.19.139.102:11222, /10.19.139.101:11222, /
10.19.139.103:11222]
Jun 24, 2014 6:00:10 AM
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory
updateServers
INFO: ISPN004014: New server added(/10.19.139.102:11222), adding to the
pool.
Jun 24, 2014 6:00:10 AM
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory
updateServers
INFO: ISPN004014: New server added(/10.19.139.101:11222), adding to the
pool.
Jun 24, 2014 6:00:10 AM org.infinispan.client.hotrod.RemoteCacheManager
start
INFO: ISPN004021: Infinispan version: 6.0.3.Final-redhat-3
stations cache size is 1086 and 1086 items are stored directly on the given
JDG node
Jun 24, 2014 6:00:10 AM org.infinispan.client.hotrod.impl.protocol.Codec10
readNewTopologyAndHash
INFO: ISPN004006: /10.19.139.103:11222 sent new topology view (id=4)
containing 3 addresses: [/10.19.139.102:11222, /10.19.139.101:11222, /
10.19.139.103:11222]
...
...
INFO: ISPN004014: New server added(/10.19.139.101:11222), adding to the
pool.
Jun 24, 2014 6:00:10 AM org.infinispan.client.hotrod.RemoteCacheManager
start
INFO: ISPN004021: Infinispan version: 6.0.3.Final-redhat-3
prices cache size is 179180 and 116816 items are stored directly on the
given JDG node
```

Note that for the replicated cache, all 1086 items are available on node 1. However for the distributed *prices* cache, only 116816 entries out of 179180 are directly available on this node. That is approximately two-thirds of the cache size, as would be expected, since the cache is configured for each entry to have two owners while there are a total of three cluster members.

4.6 Review

This reference architecture runs JBoss Data Grid servers in clustered mode. To start a JDG server in clustered mode, use the *bin/clustered.sh* script for Linux and Unix environments and the *bin/clustered.bat* script for Microsoft Windows. Configurable startup parameters for these scripts are located in the *bin/clustered.conf* file for Linux and Unix environments and the *bin/clustered.conf.bat* file for Microsoft Windows.

With the default use of the script, the server configuration file is *standalone/configuration/clustered.xml*. This file has been updated as part of this reference architecture to declare and deploy two new caches. Refer to the description of the Gas Shopper application to better understand the required cache configuration.

```
<subsystem xmlns="urn:infinispan:server:core:6.1"
                                default-cache-container="clustered">
    <cache-container name="clustered" default-cache="default"
statistics="true">
        <transport executor="infinispan-transport" lock-timeout="60000"/>
        <replicated-cache name="stations" mode="ASYNC" start="EAGER">
            <compatibility enabled="true"/>
            <locking isolation="READ_COMMITTED" acquire-timeout="30000"
                                concurrency-level="1000" striping="false"/>
            <transaction mode="NONE"/>
            <leveldb-store passivation="false" preload="true">
                <expiration path="expiration"/>
            </leveldb-store>
        </replicated-cache>
        <distributed-cache name="prices" mode="ASYNC" segments="20"
owners="2" remote-timeout="30000" start="EAGER">
            <compatibility enabled="true"/>
            <locking isolation="READ_COMMITTED" acquire-timeout="30000"
                                concurrency-level="1000" striping="true"/>
            <transaction mode="NONE"/>
            <leveldb-store passivation="false" preload="true">
                <expiration path="expiration" queue-size="200000"/>
            </leveldb-store>
        </distributed-cache>
        <distributed-cache name="default" mode="SYNC" segments="20"
owners="2" remote-timeout="30000" start="EAGER">
            <locking isolation="READ_COMMITTED" acquire-timeout="30000"
concurrency-level="1000" striping="false"/>
            <transaction mode="NONE"/>
        </distributed-cache>
        <distributed-cache name="memcachedCache" mode="SYNC" segments="20"
owners="2" remote-timeout="30000" start="EAGER">
            <locking isolation="READ_COMMITTED" acquire-timeout="30000"
concurrency-level="1000" striping="false"/>
            <transaction mode="NONE"/>
        </distributed-cache>
        <distributed-cache name="namedCache" mode="SYNC" start="EAGER"/>
    </cache-container>
    <cache-container name="security"/>
</subsystem>
```

The server is preconfigured with three distributed caches called *default*, *memcachedCache* and *namedCache*. Configure two new caches in the same section. The first cache, called stations, is a replicated cache:

```
<replicated-cache name="stations" mode="ASYNC" start="EAGER">
    <compatibility enabled="true"/>
    <locking isolation="READ_COMMITTED" acquire-timeout="30000"
                                concurrency-level="1000" striping="false"/>
    <transaction mode="NONE"/>
    <leveldb-store passivation="false" preload="true">
     <expiration path="expiration"/>
    </leveldb-store>
</replicated-cache>
```

The number of gas stations is presumed to be finite and relatively limited. Even with expansion into various global market, the fictitious *Gas Shopper* business expects the total number of gas stations in its markets to be manageable. Additionally, this data is largely immutable and not regularly updated. Contrary to price report data, gas station data is accessed by every user of the application. For these reasons as well as for demonstrative purposes, this cache has been configured as a *replicated*, and not *distributed*, cache. All the data in this cache is copied to every other node in the JDG cluster.

```
<replicated-cache name="stations" mode="ASYNC" start="EAGER"...
```

The cache is called *stations* and must be referenced by this name. It is configured to start up in *eager* mode, which initiates the cache upon server startup and contrary to *lazy* mode, does not wait for the first client hit before doing so. The replication of the cache is configured to happen asynchronously. It is assumed that for the most part, the cache would be populated soon after startup and rarely change afterwards. As such, concurrency is not a major concern.

```
    <compatibility enabled="true"/>
```

This cache is configured to run in compatibility mode. JBoss Data Grid supports various access protocols including REST, Hot Rod and Memcached. By default, each protocol stores data in the most efficient format for that protocol, ensuring transformations are not required when retrieving entries. When this data is required to be accessed from multiple protocols, compatibility mode must be enabled on caches that are being shared.[32] While Hot Rod is used as the main access protocol in this reference architecture, data access using the REST API over HTTP is also demonstrated and as such, compatibility mode is required.

```
    <locking isolation="READ_COMMITTED" acquire-timeout="30000" concurrency-
level="100" striping="false"/>
```

The *stations* cache is configured with an isolation level of *READ_COMMITED*. JBoss Data Grid only supports the READ_COMMITTED isolation level in remote client-server mode. In library mode, another option is REPEATABLE_READ, which is only used for higher levels of concurrency requirements, preserving the value of a row before a modification occurs, so that non-repeatable reads are not possible.[33]

32 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Developer_Guide/Using_Compatibility_Mode.html

33 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/About_READ_COMMITTED.html

When a client call to the JDG server results in a cache update, a clustered cache first locks the cache entry to prevent simultaneous changes by other clients, threads or on other servers. If an update cannot be made because the entry is already locked, presumably due to another simultaneous update taking place, the server waits for a determined amount of time before giving up and returning an error to the caller. This timeout period is set to 30 seconds for the *stations* cache.

The default locking behavior in JBoss Data Grid is to create a new lock for each cache entry that is updated. This high level of granularity provides better concurrency by avoiding the locking of neighbor entries as part of a segment or page. Optionally, *striping* may be used to divide the cache into a fixed number of partitions and dedicate a lock to each one of them. When *striping* is used, *concurrency level* determines the total number of locks that are created. Concurrency level also determines other low-level details of the cache set up that reflect the expected level of concurrency in the application. As previously stated, the *stations* cache is expected to largely be read-only and therefore, striping is turned off to avoid the associated overheard since little to no locking is expected at production time.

```
    <transaction mode="NONE"/>
```

JBoss Data Grid 6.3 does not support transaction mode in remote client-server use so transaction mode is set to none for clarity.

```
    <leveldb-store passivation="false" preload="true">
       <expiration path="expiration"/>
    </leveldb-store>
```

The cache is configured to use the high-performance leveldb cache store. The cache store is set up with passivation turned off, meaning that all cache contents will be persisted and not only the overflow. Additionally, the store is set to be preloaded into the cache. This means that upon startup, the cache reads and loads all the content of the cache store and will not need to read from the disk in response to every request. Finally, the expiration path simply helps determine the name of the folder where the expiration files for the cache store are stored.

```
<distributed-cache name="prices" mode="ASYNC" segments="20" owners="2"
                                        remote-timeout="30000" start="EAGER">
    <compatibility enabled="true"/>
    <locking isolation="READ_COMMITTED" acquire-timeout="30000"
                                  concurrency-level="1000" striping="true"/>
    <transaction mode="NONE"/>
    <leveldb-store passivation="false" preload="true">
         <expiration path="expiration" queue-size="200000"/>
    </leveldb-store>
</distributed-cache>
```

For each gas station, a growing number of users will provide frequent price reports. This cache is the primary driver of the use of JBoss Data Grid for this application.

```
<distributed-cache name="prices" mode="ASYNC" segments="30" owners="2"
                                        remote-timeout="30000" start="EAGER">
```

The *prices* cache is declared as a *distributed* cache with 2 owners to provide the necessary redundancy, even without a cache store. While the number of JDG nodes in the cluster can grow as the application expands into other markets and attracts more users, the memory use for the same data does not increase with the cluster size as there will always be at most two copies of each object in the cluster.

Once again, this cache is also configured to start up in *eager* mode, which initiates the cache upon server startup and contrary to *lazy* mode, does not wait for the first client hit before doing so. The replication of cache entries to a second owner is configured to happen asynchronously. While frequent updates from users may be expected, gas prices at a station do not change every minute or second so there is no urgency associated with the propagation of new data and asynchronous transfer is acceptable.

The segments parameter specifies the number of hash space segments per cluster. The recommended value for this parameter is ten multiplied by the cluster size and the default value is 80. This reference environment includes 3 nodes so 30 segments is an appropriate setting.

```
    <compatibility enabled="true"/>
```

This cache is configured to run in compatibility mode. JBoss Data Grid supports various access protocols including REST, Hot Rod and Memcached. By default, each protocol stores data in the most efficient format for that protocol, ensuring transformations are not required when retrieving entries. When this data is required to be accessed from multiple protocols, compatibility mode must be enabled on caches that are being shared.[34] While Hot Rod is used as the main access protocol in this reference architecture, data access using the REST API over HTTP is also demonstrated and as such, compatibility mode is required.

```
    <locking isolation="READ_COMMITTED" acquire-timeout="30000" concurrency-
level="1000" striping="true"/>
```

Much like the *stations* cache, the *prices* cache is also configured with an isolation level of *READ_COMMITED* and the lock timeout period is set to 30 seconds.

The prices cache is expected to contain a lot of data and have relatively frequent updates. The *striping* feature of JDG is enabled for this cache with an initial concurrency level of 1000 that may need to be reviewed and increased when the number of application users crosses certain thresholds. This configuration creates 1000 locks and divides cache entries by this number so that each lock is associated with a number of entries. Attempts to simultaneously update cache entries that are tied to the same lock would place one request on hold while the other completes, but frequent lock creation is avoided and with fine tuning the *concurrency level* based on actual load, a much better throughput can be achieved.

```
    <transaction mode="NONE"/>
```

Once again, JBoss Data Grid 6.3 does not support transaction mode in remote client-server usage mode.

```
    <leveldb-store passivation="false" preload="true">
       <expiration path="expiration"/>
    </leveldb-store>
```

This cache is also configured to use the high-performance leveldb cache store. The cache store is set up with passivation turned off, meaning that all cache contents will be persisted and not only the overflow. Additionally, the store is set to be preloaded into the cache. This means that upon startup, the cache reads and loads all the content of the cache store and will not need to read from the disk in response to every request. Finally, the expiration path simply helps determine the name of the folder where the expiration files for the cache store are stored.

34 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Developer_Guide/Using_Compatibility_Mode.html

Also modify the socket binding for the Hot Rod protocol to bind to the public interface:

```
<socket-binding-group name="standard-sockets" default-interface="public"
                      port-offset="${jboss.socket.binding.port-offset:0}">
...

  <socket-binding name="hotrod" interface="public" port="11222"/>
```

This socket binding is configured on the management interface by default, which makes it unavailable from remote servers.

5 Design and Development

5.1 Gas Shopper

The *Gas Shopper* application uses a replicated cache called *stations* to store gas station information, organized by location. This data is largely static, of limited size and quantity, and replicated to all nodes of the cluster. To easily find gas stations in a given area, a grid is envisioned where the geographical coordinates of the center of each cell is used as a cache key that holds a list of gas stations as its value.

A distributed cache called *prices* is used to store the crowd-sourced gas prices, correlated with the gas stations through a unique ID that is generated for each gas station as it's inserted in its respective cache. Gas station price reports are stored as a linked list, where the ID of the gas station is always the key to its last price report in the *prices* cache. That price report, in turn, contains the cache key of the next most recent price report for the same station and so on. This mechanism allows a new price report to be inserted by simply updating the key of the previous recent report before inserting the new one. When the cache expiration policy evicts an entry, the chronological order of the linked list means that it will always be the last entry in the linked list and can therefore be safely removed without breaking the structure.

A standalone Java client is provided to load data into the caches. The application code has been adapted to the CSV data files that can be downloaded from Esso's website. The gas station data is read and parsed into Java objects that are then sorted geographically and associated with a grid cell. The client code also creates 10 random price reports for each gas station and stores them as separate entries in the *prices* cache.

The *Gas Shopper* Web Application requests users' location from their browser and finds all geographical grid cells within a 20 miles radius of the user. The coordinates of the centers of each grid cell is used to get the *stations* cache and obtain the list of gas stations in the cell. The universally unique identifier (UUID) of each gas station is used on the *prices* cache to retrieve the most recent price report for that station.

The web application also allows users to look at the full available price history of a gas station, where the linked list is followed until the end or the point at which an entry has been evicted. There is also a link to enter a price report for a gas station.

5.2 Data Model

5.2.1 Structure

The two main custom types used in the *Gas Shopper* application are *GasStation* and *PriceReport*. Another important data type is *Coordinates*, which is used to represent a location. Instances of this class represent a geographical location, including the location of a gas station, the user's location and centers of each grid cell. Convenience methods are provided to calculate the distance between any two coordinates or find all cell centers within a given distance of a point.

5.2.1.1 Coordinates

```
public class Coordinates implements Serializable
{

    private static final long serialVersionUID = 1L;
    private double latitude;
    private double longitude;
    private static final double R = 3959.87; // In miles

    public Coordinates(double latitude, double longitude)
    {
          this.latitude = latitude;
          this.longitude = longitude;
    }

    public Coordinates(String latitude, String longitude)
                                        throws NumberFormatException
    {
          this.latitude = Double.parseDouble( latitude );
          this.longitude = Double.parseDouble( longitude );
    }
```

Geographical coordinates are identified by a latitude and a longitude and those are the only member fields of the *Coordinates* class. For caller convenience, a secondary constructor accepts latitude and longitude in string form and performs the necessary conversion.

Simple getter methods are implemented:

```
    public double getLatitude()
    {
          return latitude;
    }

    public double getLongitude()
    {
          return longitude;
    }
```

The *Coordinates* class also includes a convenience method to calculate its distance from another *Coordinates* object:

```
    public double distance(Coordinates coordinates)
    {
          double dLat = Math.toRadians( coordinates.latitude - latitude );
          double dLon = Math.toRadians( coordinates.longitude - longitude );
          double lat1 = Math.toRadians( latitude );
          double lat2 = Math.toRadians( coordinates.latitude );

          double a = Math.sin( dLat / 2 ) * Math.sin( dLat / 2 )
                           + Math.sin( dLon / 2 ) * Math.sin( dLon / 2 )
                           * Math.cos( lat1 ) * Math.cos( lat2 );
          double c = 2 * Math.asin( Math.sqrt( a ) );
          return R * c;
    }
```

This application uses all coordinates with a whole integer for both latitude and longitude as a cell center and the squares between them as grid cells. Based on this convention, the closest cell center for any given coordinate can be easily calculated:

```
    public Coordinates calculateCellCenter()
    {
          double markerLatitude = Math.round( latitude );
          double markerLongitude = Math.round( longitude );
          return new Coordinates( markerLatitude, markerLongitude );
    }
```

The combination of the two above methods can be used to find all overlapping grid cells within a given radius:

```
    public Collection<Coordinates> calculateApplicableCellCenters(double
radius)
    {
          Coordinates closestCellBorder = new
Coordinates( Math.floor( latitude ) + 0.5, Math.floor( longitude ) + 0.5 );
          if( distance( closestCellBorder ) > radius )
          {
                //search area is a single cell
                Collection<Coordinates> cellCenters = new
ArrayList<Coordinates>();
                cellCenters.add( calculateCellCenter() );
                return cellCenters;
          }
          else
          {
                Set<Coordinates> cellBorders = new HashSet<Coordinates>();
                double firstLatitude = closestCellBorder.latitude;
                double firstLongitude = closestCellBorder.longitude;
                for( int latitudeIndex = 0;; latitudeIndex++ )
                {
                      int outerLoopSize = cellBorders.size();
                      for( int longitudeIndex = 0;; longitudeIndex++ )
                      {
```

```
                        int innerLoopSize = cellBorders.size();
                        addIfWithinRadius( cellBorders, new
Coordinates( firstLatitude + latitudeIndex, firstLongitude +
longitudeIndex ), radius );
                        addIfWithinRadius( cellBorders, new
Coordinates( firstLatitude + latitudeIndex, firstLongitude -
longitudeIndex ), radius );
                        addIfWithinRadius( cellBorders, new
Coordinates( firstLatitude - latitudeIndex, firstLongitude +
longitudeIndex ), radius );
                        addIfWithinRadius( cellBorders, new
Coordinates( firstLatitude - latitudeIndex, firstLongitude -
longitudeIndex ), radius );
                        if( cellBorders.size() == innerLoopSize )
                        {
                              break;
                        }
                  }
                  if( cellBorders.size() == outerLoopSize )
                  {
                        break;
                  }
            }
            Set<Coordinates> cellCenters = new HashSet<Coordinates>();
            for( Coordinates cellBorder : cellBorders )
            {
                  cellCenters.add( new
Coordinates( Math.floor( cellBorder.latitude ),
Math.floor( cellBorder.longitude ) ) );
                  cellCenters.add( new
Coordinates( Math.floor( cellBorder.latitude ),
Math.ceil( cellBorder.longitude ) ) );
                  cellCenters.add( new
Coordinates( Math.ceil( cellBorder.latitude ),
Math.floor( cellBorder.longitude ) ) );
                  cellCenters.add( new
Coordinates( Math.ceil( cellBorder.latitude ),
Math.ceil( cellBorder.longitude ) ) );
            }
            return cellCenters;
      }
  }

  private void addIfWithinRadius(Set<Coordinates> cellBorders, Coordinates
candidate, double radius)
  {
      if( distance( candidate ) <= radius )
      {
            cellBorders.add( candidate );
      }
  }
```

When a geographical coordinate is used as a cache key, it is easier and more convenient to use a string representation of the coordinates instead:

```
    public String asString()
    {
          return "(" + latitude + "," + longitude + ")";
    }
```

If needed, the string conversion is easily reversible:

```
    public static Coordinates fromString(String coordinateString)
    {
          int separatorIndex = coordinateString.indexOf( ',' );
          String latitude = coordinateString.substring( 1, separatorIndex );
          String longitude = coordinateString.substring
                ( separatorIndex + 1, coordinateString.indexOf( ')' ) );
          Coordinates coordinates = new Coordinates( latitude, longitude );
          return coordinates;
    }
```

The *toString()* method returns the full content of the gas station object to help logging:

```
    @Override
    public String toString()
    {
          return "Coordinates [latitude=" + latitude + ", longitude="
                                                      + longitude + "]";
    }
```

5.2.1.2 GasStation

```
public class GasStation implements Serializable
{

    private static final long serialVersionUID = 1L;
    private String company;
    private String address;
    private String telephone;
    private Coordinates coordinates;
    private UUID stationId;

    public GasStation(String company, String address, String telephone,
Coordinates coordinates)
    {
          this.company = company;
          this.address = address;
          this.telephone = telephone;
          this.coordinates = coordinates;
          this.stationId = UUID.randomUUID();
    }
```

Each gas station is identified by a unique identifier that is generated randomly upon creation of the object. Other fields include the gas station company / brand, its address, telephone number and location. All these fields must be provided to the constructor when instantiating the class and as a result, no setter method is required for these fields.

Simple getter methods are implemented:

```
    public String getCompany()
    {
          return company;
    }

    public String getAddress()
    {
          return address;
    }

    public String getTelephone()
    {
          return telephone;
    }

    public Coordinates getCoordinates()
    {
          return coordinates;
    }

    public UUID getStationId()
    {
          return stationId;
    }
```

The *toString()* method returns the full content of the *GasStation* object to help logging:

```
    @Override
    public String toString()
    {
          return "GasStation [company=" + company + ", address=" + address
                      + ", telephone=" + telephone + ", coordinates="
                      + coordinates + ", stationId=" + stationId + "]";
    }
```

5.2.1.3 PriceReport

```
public class PriceReport implements Serializable
{

    private static final long serialVersionUID = 1L;
    private String ipAddress;
    private Date date;
    private float regular;
    private float midgrade;
    private float premium;
    private float diesel;
    private String lastReportKey;
```

```
    public PriceReport(String ipAddress, float regular, float midgrade,
                                              float premium, float diesel)
    {
          this.ipAddress = ipAddress;
          this.regular = regular;
          this.midgrade = midgrade;
          this.premium = premium;
          this.diesel = diesel;
          this.date = new Date();
    }
```

The most recent price report for a gas station is always stored in the cache under a key matching the UUID of the gas station. The only way to locate and access earlier price reports is by following the linked list. The *lastReportKey* field contains the cache key of the next most recent price report.

Simple getter methods are implemented:

```
    public Date getDate()
    {
          return date;
    }

    public void setDate(Date date)
    {
          this.date = date;
    }

    public String getIpAddress()
    {
          return ipAddress;
    }

    public float getRegular()
    {
          return regular;
    }

    public float getMidgrade()
    {
          return midgrade;
    }

    public float getPremium()
    {
          return premium;
    }

    public float getDiesel()
    {
          return diesel;
    }

    public String getLastReportKey()
    {
```

```
        return lastReportKey;
    }
```

The only mutable part of the class is the cache key of the next price report:

```
    public void setLastReportKey(String lastReportKey)
    {
        this.lastReportKey = lastReportKey;
    }
```

Once again, implementing the *toString()* method returns the full content of the object to help logging:

```
    @Override
    public String toString()
    {
        return "PriceReport [ipAddress=" + ipAddress + ", date=" + date
            + ", regular=" + regular + ", midgrade=" + midgrade
            + ", premium=" + premium + ", diesel=" + diesel + "]";
    }
```

5.2.2 Sample Data

As described in the Downloads, comma-separated values files may be downloaded and used to populate the cache with real gas station data. The standalone client code targets the CSV format of publicly available source of data mentioned in this reference architecture document but with minor changes, it can easily be adapted to read other formats as well.

5.2.2.1 Gas Stations

The standalone client code is able to parse the comma-separated values in the CSV files and create a GasStation object for each line. A random UUID is generated for each gas station as the object is instantiated. These objects are populated in the *stations* cache in the appropriate format.

The standalone client expects a command-line argument to indicate the operation that is to be performed. To load the sample data, provide load as the first argument followed by the name of one or multiple CSV files that contain the gas station data:

```
public class Client
{

    public static void main(String[] args) throws Exception
    {
        Client loader = new Client();
        if( args.length > 0 )
        {
            if( args[0].equals( "load" ) )
            {
                Collection<String> gasStations =
                            loader.cacheGasStations( args[1] );
                loader.cachePriceReports( gasStations );
            }
```

The first action is to cache the gas stations:

```
Collection<String> gasStations = loader.cacheGasStations( args[1] );
```

This results in a call to a method withing the same class:

```
private Collection<String> cacheGasStations(String filenames) throws
Exception
{
     Map<String, Map<UUID, GasStation>> gasStations =
readGasStations( filenames );
     System.out.println( "Will store gas station info as "
                          + gasStations.size() + " cache entries" );
     getCache( "stations" ).putAll( gasStations, -1, TimeUnit.DAYS );
     Collection<String> stationIds = new ArrayList<String>();
     for( Map<UUID, GasStation> map : gasStations.values() )
     {
           Set<UUID> keys = map.keySet();
           for( UUID uuid : keys )
           {
                 stationIds.add( uuid.toString() );
           }
     }
     return stationIds;
}
```

The first line of the method calls another method of the same class to read and parse the files, before returning them as a *Map* organized by geographical location:

```
private Map<String, Map<UUID, GasStation>> readGasStations(String filenames)
throws Exception
{
     List<String[]> stations = new ArrayList<String[]>();
     String[] csvFiles = filenames.split( "," );
     for( String csvFile : csvFiles )
     {
           CSVReader csvReader = new CSVReader( new InputStreamReader( new
FileInputStream( csvFile ) ) );
           stations.addAll( csvReader.readAll() );
           csvReader.close();
     }
     System.out.println( "Read " + stations.size() + " stations from file" );
     Map<String, Map<UUID, GasStation>> gasStations = new HashMap<String,
Map<UUID, GasStation>>();
     for( String[] stationDetails : stations )
     {
           Coordinates coordinates = new Coordinates( stationDetails[1],
stationDetails[0] );
           String[] addressParts = stationDetails[3].split( "\n" );
           String address = addressParts[0] + ", " + addressParts[1];
           String phone = addressParts[2];
           GasStation gasStation = new GasStation( stationDetails[2],
address, phone, coordinates );
           Map<UUID, GasStation> cellStations =
```

```
gasStations.get( coordinates.calculateCellCenter().asString() );
          if( cellStations == null )
          {
                cellStations = new HashMap<UUID, GasStation>();

    gasStations.put( coordinates.calculateCellCenter().asString(),
cellStations );
          }
          cellStations.put( gasStation.getStationId(), gasStation );
    }

    return gasStations;
}
```

The filenames argument may be a single file or the path to multiple files in comma-separated form. Using split with a regular expression returns an array where each element is exactly one file:

```
List<String[]> stations = new ArrayList<String[]>();
String[] csvFiles = filenames.split( "," );
```

A simple loop over the files allows the client code to use the **OpenCSV** library to read each line as a *String Array* and store all of them in a single list:

```
for( String csvFile : csvFiles )
{
    CSVReader csvReader = new CSVReader( new InputStreamReader( new
FileInputStream( csvFile ) ) );
    stations.addAll( csvReader.readAll() );
    csvReader.close();
}
```

At this point the list contains one entry per gas station, although the gas station data itself has not yet been parsed and remains in comma-separated format:

```
System.out.println( "Read " + stations.size() + " stations from file" );
```

The covered geography is treated as a grid and divided into cells where coordinates with whole numbers as both latitude and longitude are designated cell centers. For example, the location with a latitude of 34.0 and a longitude of -118.0 is a cell center for a cell that covers all points where their latitude is smaller than 34.5 and equal or larger than 33.5 while their longitude is smaller than -117.5 and equal or later than -118.5. As a result, the map created by the client code, like the cache itself, has a limited number coordinates as keys with only whole integers as their latitude and longitude. Each such coordinate has a map of Gas Station objects as its value, arranged by the station UUID:

```
Map<String, Map<UUID, GasStation>> gasStations = new HashMap<String,
Map<UUID, GasStation>>();
for( String[] stationDetails : stations )
{
    Coordinates coordinates = new Coordinates( stationDetails[1],
stationDetails[0] );
```

```
    String[] addressParts = stationDetails[3].split( "\n" );
    String address = addressParts[0] + ", " + addressParts[1];
    String phone = addressParts[2];
    GasStation gasStation = new GasStation( stationDetails[2], address,
phone, coordinates );
    Map<UUID, GasStation> cellStations =
gasStations.get( coordinates.calculateCellCenter().asString() );
    if( cellStations == null )
    {
          cellStations = new HashMap<UUID, GasStation>();
          gasStations.put( coordinates.calculateCellCenter().asString(),
cellStations );
    }
    cellStations.put( gasStation.getStationId(), gasStation );
}
```

Once the map of gas stations is created and returned, it is directly stored in the cache with a value of -1 indicating no expiration:

```
getCache( "stations" ).putAll( gasStations, -1, TimeUnit.DAYS );
```

Finally, a list of station IDs is prepared and returned to the caller for confirmation and subsequent use:

```
    Collection<String> stationIds = new ArrayList<String>();
    for( Map<UUID, GasStation> map : gasStations.values() )
    {
          Set<UUID> keys = map.keySet();
          for( UUID uuid : keys )
          {
                stationIds.add( uuid.toString() );
          }
    }
    return stationIds;
```

The cache, used to store the gas station info, is retrieved using a convenience method that uses the Hot Rod library:

```
private static RemoteCache<String, Object> getCache(String cacheName) throws
IOException
{
    Properties properties = new Properties();

properties.load( Client.class.getResourceAsStream( "/jdg.properties" ) );
    ConfigurationBuilder builder = new ConfigurationBuilder();
    String host = properties.getProperty( "jdg.host" );
    String port = properties.getProperty( "jdg.port" );
    //System properties override the defaults in our property files:
    host = System.getProperty( "jdg.host", host );
    port = System.getProperty( "jdg.port", port );
    builder.addServer().host( host ).port( Integer.valueOf( port ) );
    RemoteCacheManager cacheManager = new
RemoteCacheManager( builder.build() );
```

```
    RemoteCache<String, Object> cache = cacheManager.getCache( cacheName );
    return cache;
}
```

The client code bundles property files with default values for JDG host and specially port, but looks to provided system properties and gives those priority.

5.2.2.2 Price Reports

After populating the gas stations, the client code proceeds to generate 10 random price reports for each gas station and stores them in the *prices* cache.

The list of station Ids is passed to the method that generates and stores the price reports:

```
loader.cachePriceReports( gasStations );
```

This method calls another method to generate the price reports and then stores them in the cache with an expiration lifespan of 24 hours.

The method to create price reports is more flexible than the default client behavior. In calling it, the client code hardcodes $4.19 as the average gas price for regular gas and that exactly 10 price repots should be generated for each gas station:

```
private void cachePriceReports(Collection<String> stationIds) throws
Exception
{
    Map<String, PriceReport> priceReports =
                          generatePriceReports( stationIds, 10, 4.19f );

    System.out.println( "Will store price reports as "
                          + priceReports.size() + " cache entries" );

    getCache( "prices" ).putAll( priceReports, 1, TimeUnit.DAYS );
}
```

The price report generation method distributes the price reports uniformly over the last 24 hours:

```
private static Map<String, PriceReport>
generatePriceReports(Collection<String> stationIds, int reportPerStation,
float basePrice)
{
    Date[] reportDates = new Date[reportPerStation];
    long interval = 12 * 60 * 60000 / ( reportPerStation - 1 );
    long millis = System.currentTimeMillis() - 12 * 60 * 60000;
    for( int index = 0; index < reportDates.length; index++ )
    {
          reportDates[index] = new Date( millis );
          millis += interval;
    }
```

Price reports are stored in a map in the same format as the *prices* cache:

```
Map<String, PriceReport> priceReports = new HashMap<String, PriceReport>();
```

For each gas station, the configured number of reports, in this case hardcoded to 10, is generated:

```
for( String stationId : stationIds )
{
```

The PriceGenerator class is called to generate random prices based on the provided average price and a maximum deviation that is hardcoded to $1.20:

```
for( int index = 0; index < reportDates.length; index++ )
{
    PriceReport priceReport =
PriceReportGenerator.getRandomPriceReport( basePrice, 1.20f );
    priceReport.setDate( reportDates[index] );
```

If this is not the last (least recent) price report, it should also contain the cache key to locate the previous price report. As explained in the description of Gas Shopper, the price reports are stored as a linked list where the cache key to the first entry, the most recent price of gas at a given gas station, is the station ID. That price report then points to the previous report and so on:

```
if( index > 0 )
{
    priceReport.setLastReportKey(
                    stationId + "_" + reportDates[index - 1].getTime() );
}
```

The key of the entry, when it's not the most recent price report, has its timestamp appended to the station ID:

```
String key;
if( index == reportDates.length - 1 )
{
    key = stationId;
}
else
{
    key = stationId + "_" + reportDates[index].getTime();
}
priceReports.put( key, priceReport );
```

Generating the mockup price reports is fairly simple. It relies heavily on the *Random* class provided in Java's *Math* library:

```
public class PriceReportGenerator
{

    private static Random random = new Random( System.currentTimeMillis() );
```

A random base price is generated for regular gasoline and the price of diesel, midgrade and premium gas is derived from that value:

```
public static PriceReport getRandomPriceReport( float baseline, float
variance )
{
     int randomMax = ((int)(variance * 10) * 2 + 1);
     int randomInt = random.nextInt( randomMax );
     float adjustedInt = randomInt - ((randomMax - 1) / 2);
     float deviation = adjustedInt / 10;
     float regular = baseline + deviation;
     regular = new BigDecimal( regular ).setScale( 2,
RoundingMode.HALF_UP ).floatValue();
     float midgrade = new BigDecimal( regular + 0.1 ).setScale( 2,
RoundingMode.HALF_UP ).floatValue();
     float premium = new BigDecimal( regular + 0.2 ).setScale( 2,
RoundingMode.HALF_UP ).floatValue();
     float diesel = new BigDecimal( regular - 0.1 ).setScale( 2,
RoundingMode.HALF_UP ).floatValue();
     PriceReport priceReport = new PriceReport( getRandomIP(), regular,
midgrade, premium, diesel );
     return priceReport;
}
```

Random IP addresses are generated to represent the fictional user who reported each price:

```
private static String getRandomIP()
{
     int first = random.nextInt( 251 ) + 1;
     if( first >= 192 )
     {
          first += 3;
     }
     else if( first >= 172 )
     {
          first += 2;
     }
     else if( first >= 10 )
     {
          first++;
     }
     int second = random.nextInt( 256 );
     int third = random.nextInt( 256 );
     int fourth = random.nextInt( 255 ) + 1;
     return first + "." + second + "." + third + "." + fourth;
}
```

5.3 Web Application

The GasShopperWeb application is a standard Java EE web application that is designed and developed to demonstrate the use of JBoss Data Grid in remote client-server mode. This application allows users to get a list of gas stations close by, sorted by distance from the user location, along with gas prices at each station. Users can also provide price updates by reporting new prices for a gas station. The price history of a gas station may also be queried to the extent that it is still available. Entries are stored in the *prices* cache with an expiration lifespan of 1 day so historical prices will be limited to 24 hours.

The web application complies with the Servlet 3.0 Specification and uses annotations instead of a *web.xml* deployment descriptor:

```
@WebServlet(urlPatterns = {"/*"})
public class Servlet extends HttpServlet
{

    private static final long serialVersionUID = 1L;
```

Given the frequent interaction with both *stations* and *prices* caches, a remote cache reference to each is constructed and held as soon as the Servlet is instantiated:

```
    private RemoteCache<String, Map<UUID, GasStation>> gasStationCache
                                                = getGasStationCache();
    private RemoteCache<String, PriceReport> priceReportCache
                                                = getPriceReportCache();
...

    private RemoteCache<String, Map<UUID, GasStation>> getGasStationCache()
    {
          try
          {
                RemoteCacheManager cacheManager = getRemoteCacheManager();
                return cacheManager.getCache( "stations" );
          }
          catch( IOException e )
          {
                throw new IllegalStateException( e );
          }
    }

    private RemoteCache<String, PriceReport> getPriceReportCache()
    {
          try
          {
                RemoteCacheManager cacheManager = getRemoteCacheManager();
                return cacheManager.getCache( "prices" );
          }
          catch( IOException e )
          {
                throw new IllegalStateException( e );
          }
    }
```

The web application includes a jdg.properties files in its resources with a default hostname and port to contact a JDG server through Hot Rod:

```
jdg.host=192.168.1.5
jdg.port=11222
```

The remote cache manager is constructed by using these properties as default, but replacing them with any system properties that might have been set up in the container:

```
private RemoteCacheManager getRemoteCacheManager() throws IOException
{
    Properties properties = new Properties();
    properties.load( getClass().getResourceAsStream( "/jdg.properties" ) );
    ConfigurationBuilder builder = new ConfigurationBuilder();
    String host = properties.getProperty( "jdg.host" );
    String port = properties.getProperty( "jdg.port" );
    //System properties override the defaults in our property files:
    host = System.getProperty( "jdg.host", host );
    port = System.getProperty( "jdg.port", port );
    builder.addServer().host( host ).port( Integer.valueOf( port ) );
    RemoteCacheManager cacheManager
                      = new RemoteCacheManager( builder.build() );
    return cacheManager;
}
```

5.3.1 Location Detection

The approximate location of a web user may easily be queried through HTML 5 compliant JavaScript. When a user first reaches the web application, the first request is a simple HTTP GET request. The Servlet responds to this request by loading by a bundled html resource and rendering it on the caller's browser:

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    PrintWriter writer = response.getWriter();
    writer.println( "<html xmlns=\"http://www.w3.org/1999/xhtml\">" );
    writer.println( "<head/>" );
    writer.println( "<body onload=\"getLocation()\">" );
    writer.println( "<p id=\"message\"></p>" );
    writer.println( "<script>" );
    include( "/geolocation.txt", writer );
    include( "/post.txt", writer );
    writer.println( "</script>" );
    writer.println( "</body>" );
    writer.println( "</html>" );
}
```

The HTML code runs a JavaScript function as soon as it is loaded in the browser:

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head/>
<body onload="getLocation()">
     <p id="message"></p>
     <script>
```

A placeholder called *message* is reserved in the HTML for any potential errors.

The geolocation code consists of three JavaScript functions (along with one variable), which are best stored and managed outside the Java code. This code is included in a file named geolocation.txt under the web application resources. It is read by a convenience method in the Servlet and inserted in the HTML response:

```
private void include(String filename, Writer output) throws IOException
{
     Reader input = new InputStreamReader(
                              getClass().getResourceAsStream( filename ) );
     char[] buffer = new char[4096];
     int n = 0;
     while( -1 != ( n = input.read( buffer ) ) )
     {
          output.write( buffer, 0, n );
     }
}
```

The first line of the JavaScript creates a reference to the message placeholder. The getLocation() function checks to see if the user's browser support Geolocation and if it does not, displays an error message explaining the problem:

```
     var x = document.getElementById("message");
     function getLocation() {
       if (navigator.geolocation) {
          navigator.geolocation.getCurrentPosition(redirect, showError);
       } else {
          x.innerHTML = "Geolocation is not supported by this browser.";
       }
     }
```

The Geolocation feature takes two JavaScript callback methods where the first one is called after successful location detection and the second one only gets called if the location is not determined.

There may be various reasons why the location is not returned. The web user may decline to share their location with the host, the device and its environment may be unable to detect the location, the attempt to detect location may have taken too long or another type of error may have occurred. An error message corresponding to the given scenario is printed:

```
function showError(error) {
     switch (error.code) {
     case error.PERMISSION_DENIED:
          x.innerHTML = "User denied the request for Geolocation."
          break;
     case error.POSITION_UNAVAILABLE:
```

```
            x.innerHTML = "Location information is unavailable."
            break;
      case error.TIMEOUT:
            x.innerHTML = "The request to get user location timed out."
            break;
      case error.UNKNOWN_ERROR:
            x.innerHTML = "An unknown error occurred."
            break;
      }
}
```

In the case of successful retrieval of the user's location, the redirect method posts the user's latitude and longitude back to the Servlet:

```
function redirect(position) {
      postSameWindow('.', {
            latitude : position.coords.latitude,
            longitude : position.coords.longitude,
            gasStations : ''
      })
}
```

To post a request, the code relies on a separate JavaScript file called post.txt that is reused in other scenarios. It is separately read and inserted into the response by the Servlet. The function simply takes a number of parameters and posts them to the specified URL. There are two similar functions provided where one opens the response of the POST in a new window whereas the other one uses the current browser window:

```
function postNewWindow(path, params) {
      var form = document.createElement("form");
      form.setAttribute("method", "post");
      form.setAttribute("target", "_blank");
      form.setAttribute("action", path);

      for ( var key in params) {
            if (params.hasOwnProperty(key)) {
                  var hiddenField = document.createElement("input");
                  hiddenField.setAttribute("type", "hidden");
                  hiddenField.setAttribute("name", key);
                  hiddenField.setAttribute("value", params[key]);

                  form.appendChild(hiddenField);
            }
      }

      document.body.appendChild(form);
      form.submit();
}

function postSameWindow(path, params) {
      var form = document.createElement("form");
      form.setAttribute("method", "post");
      form.setAttribute("target", "_self");
```

```
    form.setAttribute("action", path);

    for ( var key in params) {
          if (params.hasOwnProperty(key)) {
                var hiddenField = document.createElement("input");
                hiddenField.setAttribute("type", "hidden");
                hiddenField.setAttribute("name", key);
                hiddenField.setAttribute("value", params[key]);

                form.appendChild(hiddenField);
          }
    }

    document.body.appendChild(form);
    form.submit();
}
```

In this case, the address to post to is ".", which means the request is posted back to the same Servlet. In addition to the latitude and longitude, the *gasStations* parameter is also provided as a market to indicate that the response should be a list of gas stations near the user location. The Servlet looks for a marker parameter to decide how to respond to a request:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    Map<String, String[]> params = request.getParameterMap();
    if( params.containsKey( "gasStations" ) )
    {
          displayGasStations( request, response );
    }
```

5.3.2 Gas Station Lookup

Upon receiving an HTTP POST request with a parameter requesting a list of gas stations, the Servlet looks for the geographical coordinates based on the latitude and longitude parameters:

```
private void displayGasStations(HttpServletRequest request,
HttpServletResponse response) throws IOException
{
    Coordinates coordinates = new Coordinates(
                                request.getParameter( "latitude" ),
                                request.getParameter( "longitude" ) );
```

Gas stations are grouped by geographical grid cells and stored in the cache. To look up nearby gas stations, the first step is to find the grid cells you are interested in. The Coordinates class has a convenience method that returns all cells (identified by the coordinates of the cell's center) that include points within a given distance of the starting point. The Servlet calculates grid cells covering gas stations as far as 20 miles away:

```
Collection<Coordinates> cellCenters =
                      coordinates.calculateApplicableCellCenters( 20 );
```

The HTML response again inserts the post JavaScript functions for use:

```
PrintWriter writer = response.getWriter();
writer.println( "<html xmlns=\"http://www.w3.org/1999/xhtml\">" );
writer.println( "<head/>" );
writer.println( "<body>" );
writer.println( "    <script>" );
include( "/post.txt", writer );
writer.println( "    </script>" );
```

With the geographical grid cell centers calculated, the Servlet queries the cache through the Hot Rod protocol for each cell center set of coordinates and aggregates the results in a single *List*:

```
List<GasStation> allStations = new ArrayList<GasStation>();
for( Coordinates cellCenter : cellCenters )
{
    Map<UUID, GasStation> gasStationsMap =
        (Map<UUID, GasStation>)gasStationCache.get( cellCenter.asString() );
    allStations.addAll( gasStationsMap.values() );
}
```

The next step is to sort the gas stations based on their distance from the user. The *distance* method of the *Coordinates* class helps achieve this. The gas station data is stored in a TreeMap to easily sort the content based on the natural order of the map key; the distance from the user is the map key and the gas station object is the value:

```
Map<Double, GasStation> sortedStations = new TreeMap<Double, GasStation>();
for( GasStation gasStation : allStations )
{
    double distance = coordinates.distance( gasStation.getCoordinates() );
    sortedStations.put( distance, gasStation );
}
```

The distance is displayed with a maximum of one decimal places using a NumberFormat implementation:

```
DecimalFormat decimalFormat = new DecimalFormat( "###.# miles" );
```

At this point, the gas stations can be rendered in the HTML response:

```
for( Entry<Double, GasStation> entry : sortedStations.entrySet() )
{

writer.println( "-------------------------------------------------------------
-------------------------------------------------------------" );
    writer.println( "<br/>" );
    writer.println( entry.getValue().getCompany() );
    writer.println( "<br/>" );
```

After the separator, the first line printed is the gas station brand and store name. The second line is the most recent available price report for the gas station, which is the presumed current price of fuel at the location. This information is queried from the *prices* cache and formatted and printed with the help of a different method:

```
    PriceReport priceReport =
priceReportCache.get( entry.getValue().getStationId().toString() );
    writer.println( getLinkedPriceReport(
                     entry.getValue().getStationId(), priceReport ) );
```

The getLinkedPriceReport() method prints the price and provides links to view the full price history, or report updated prices.

To view full price history, the JavaScript posts a message to the Servlet in a new window, providing *stationId* and the *history* marker. This is done in a new window or tab so that it can be easily closed to get back to the main list.

To report prices for a gas station, a message is posted on the current window, once again providing *stationId* but this time with a market of *report*:

```
private String getLinkedPriceReport(UUID uuid, PriceReport priceReport)
{
    StringWriter stringWriter = new StringWriter();
    stringWriter.append( String.valueOf( priceReport.getRegular() ) );
    stringWriter.append( " / " );
    stringWriter.append( String.valueOf( priceReport.getMidgrade() ) );
    stringWriter.append( " / " );
    stringWriter.append( String.valueOf( priceReport.getPremium() ) );
    stringWriter.append( " / " );
    stringWriter.append( String.valueOf( priceReport.getDiesel() ) );
    stringWriter.append( " - " );
    stringWriter.append( "<a href=\"javascript:void(0)\"
                          onclick=\"postNewWindow('.', {stationId : '" );
    stringWriter.append( uuid.toString() );
    stringWriter.append( "', history : ''});\">" );
    stringWriter.append( "Price History" );
    stringWriter.append( "</a>" );
    stringWriter.append( " - " );
    stringWriter.append( "<a href=\"javascript:void(0)\"
                          onclick=\"postSameWindow('.', {stationId : '" );
    stringWriter.append( uuid.toString() );
    stringWriter.append( "', report : ''});\">" );
    stringWriter.append( "Report Prices" );
    stringWriter.append( "</a>" );
    return stringWriter.toString();
}
```

The distance is printed next, using the formatter object:

```
    writer.println( "<br/>" );
    writer.println( decimalFormat.format( entry.getKey() ) );
    writer.println( "<br/>" );
```

Finally, the address and phone number of the gas station are printed on the next two lines:

```
    writer.println( getLinkedAddress( entry.getValue().getAddress() ) );
    writer.println( "<br/>" );
    writer.println( entry.getValue().getTelephone() );
    writer.println( "<p/>" );
}
```

The address is returned as an HTML link that uses **`Google Maps`** to give directions from the detected user location to the gas station in question:

```
private String getLinkedAddress(Coordinates coordinates, String address)
{
    StringWriter stringWriter = new StringWriter();
    String source = coordinates.getLatitude() + ","
                                + coordinates.getLongitude();
    stringWriter.append( "<a href=\"http://maps.google.com/maps?saddr="
                                    + source + "&daddr=" + address );
    stringWriter.append( "\" target=\"_blank\">" );
    stringWriter.append( address );
    stringWriter.append( "</a>" );
    return stringWriter.toString();
}
```

5.3.3 Price History Lookup

The Servlet looks for a marker called *history* to detect that the price history is being queried:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    ...
    else if( params.containsKey( "history" ) )
    {
         displayPriceHistory( request, response );
    }
```

The *displayPriceHistory* method prints the web response:

```
private void displayPriceHistory(HttpServletRequest request,
HttpServletResponse response) throws IOException
{
    PrintWriter writer = response.getWriter();
    writer.println( "<html
xmlns=\"http://www.w3.org/1999/xhtml\"><HEAD/><BODY>" );
    writer.println( "<head/>" );
    writer.println( "<body>" );
```

The price reports for any given gas station are stored in the cache as a linked list. The most recent price report can be retrieved from the cache by using the gas station UUID as key:

```
 priceReport = priceReportCache.get( request.getParameter( "stationId" ) );
```

The lastReportKey field of this price report acts as the cache key to the previous one and so on, until either the lastReportKey is null because no older price report ever existed, or the object retrieved from the cache is null because it has expired:

```
while( priceReport != null )
{

writer.println( "------------------------------------------------------------
------------------------------------------------------------" );
    writer.println( "<br/>" );
    writer.append( "Reported on " );
    writer.println( priceReport.getDate() );
    writer.println( "<br/>" );
    writer.append( "By user " );
    writer.println( priceReport.getIpAddress() );
    writer.println( "<br/>" );
    writer.append( "Price: " );
    writer.append( String.valueOf( priceReport.getRegular() ) );
    writer.append( " / " );
    writer.append( String.valueOf( priceReport.getMidgrade() ) );
    writer.append( " / " );
    writer.append( String.valueOf( priceReport.getPremium() ) );
    writer.append( " / " );
    writer.append( String.valueOf( priceReport.getDiesel() ) );
    writer.println( "<p/>" );

    if( priceReport.getLastReportKey() == null )
    {
          break;
    }
    else
    {
          priceReport = priceReportCache.get( priceReport.getLastReportKey()
);
    }
}
writer.println( "</body>" );
writer.println( "</html>" );
```

5.3.4 Price Reporting

When the user clicks the link to report prices for a gas station, a message is posted to the Servlet with the *report* marker. This is handled by the *queryPriceReport* method:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    ...
    else if( params.containsKey( "report" ) )
    {
          queryPriceReport( request, response );
    }
```

The response returned by this method is an HTML form that collects the price of fuel in various grades and submits them back to the Servlet with a marker of *reported*:

```
private void queryPriceReport(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException
{
    PrintWriter writer = response.getWriter();
    writer.println( "<html xmlns=\"http://www.w3.org/1999/xhtml\">" );
    writer.println( "<head/>" );
    writer.println( "<body>" );
    writer.println( "<form method=\"post\" action=\"\">" );
    include( "/pricereporttable.txt", writer );
    writer.println( "<input type=\"hidden\" name=\"stationId\" value=\""
                      + request.getParameter( "stationId" ) + "\">" );
    writer.println( "<input type=\"hidden\" name=\"reported\">" );
    writer.println( "</form>" );
    writer.println( "</body>" );
    writer.println( "</html>" );
}
```

The HTML code for the form table is saved as a resources file called *pricereporttable.txt*:

```
<table cellspacing="20">
   <tr>
      <td>Regular: </td>
      <td>
         <input type="number" name="regular" step="0.001">
      </td>
   </tr>
   <tr>
      <td>Midgrade: </td>
      <td>
         <input type="number" name="midgrade" step="0.001">
      </td>
   </tr>
   <tr>
      <td>Premium: </td>
      <td>
         <input type="number" name="premium" step="0.001">
      </td>
   </tr>
   <tr>
      <td>Diesel: </td>
      <td>
         <input type="number" name="diesel" step="0.001">
      </td>
   </tr>
   <tr>
      <td>
         <input type="submit" value="Report" style="font-size:x-large">
      </td>
   </tr>
</table>
```

Once the form is submitted, the Servlet looks for the provided marker to handle it properly:

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException
{
    ...
    else if( params.containsKey( "reported" ) )
    {
          reportPrice( request, response );
    }
```

This method starts by collecting the submitted data:

```
private void reportPrice(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException
{
    String stationId = request.getParameter( "stationId" );
    Float regular = Float.valueOf( request.getParameter( "regular" ) );
    Float midgrade = Float.valueOf( request.getParameter( "midgrade" ) );
    Float premium = Float.valueOf( request.getParameter( "premium" ) );
    Float diesel = Float.valueOf( request.getParameter( "diesel" ) );
```

The IP address of the caller is also collected and a price report object is created with the available information:

```
    String ipAddress = request.getRemoteAddr();
    PriceReport thisPriceReport = new PriceReport( ipAddress, regular,
                                            midgrade, premium, diesel );
```

The previously stored price report is retrieved from the cache. This is the most recent price report up to this point but loses its status as the current report gets stored. It should now have its cache key changed to have its timestamp appended to the station ID. The new price report keeps a reference to this cache key in order to preserve the linked list structure:

```
    PriceReport lastPriceReport = priceReportCache.get( stationId );
    String lastReportKey = stationId + "_" +
                                lastPriceReport.getDate().getTime();
    thisPriceReport.setLastReportKey( lastReportKey );
```

The new price report is stored and the old one is updated, so both can be sent to the cache in a single request:

```
    Map<String, PriceReport> priceReports =
                                        new HashMap<String, PriceReport>();
    priceReports.put( stationId, thisPriceReport );
    priceReports.put( lastReportKey, lastPriceReport );
    priceReportCache.putAll( priceReports );
```

Finally, the caller is sent to the starting page where fresh data is queried from the cache:

```
    doGet( request, response );
```

5.4 REST Interface

Both the web application and the load function of the standalone client use the Hot Rod protocol to communicate with the JDG servers. Where possible the use of the Hot Rod client is preferable as it provides superior performance as well as ease of use.

Hot Rod clients are available for various languages and environments and it is preferable to use Hot Rod, due to its high performance and efficiency, whenever and wherever at all possible. However there are technical environments where support for Hot Rod is not available. In such cases, REST over HTTP provides a highly portable and standardized protocol alternative that can be easily used in various client environments.

For a JDG server running on host 10.19.139.101 and listening on HTTP port 8080, a cache value can retrieved through an HTTP GET request to:

http://10.19.139.101:8080/rest/cacheName/cacheKey

While the Hot Rod protocol is private by default, the HTTP listener is public and requires more security. For this reason, the default configuration of JBoss Data Grid server secures REST calls to caches and requires an authenticated user with the security role of *REST*. In this reference architecture, such a user is added to every JBoss Data Grid Server while configuring it.

The standalone client code provides a simple Java method to issue a REST request and receive a JSON response:

```
private static String restGet(String urlServerAddress, String username,
String password) throws IOException
{
    System.out.println( "---------------------------------------" );
    System.out.println( "Executing GET " + urlServerAddress );
    System.out.println( "---------------------------------------" );

    HttpURLConnection connection =
          (HttpURLConnection)new URL( urlServerAddress ).openConnection();
```

The default server configuration uses basic authentication, which means the username and password have to be separated by a colon and then encoded using the *Base64* scheme and included in the request headers:

```
    connection.setRequestProperty( "Authorization", "Basic " +
         Base64.encodeBytes( ( username + ":" + password ).getBytes() ) );
```

In addition to other standard request headers, the client also specifies that the response should be provided in JSON form:

```
    connection.setRequestMethod( "GET" );
    connection.setRequestProperty( "Content-Type", "text/plain" );
    connection.setRequestProperty( "Accept", "application/json" );
    connection.setDoOutput( true );
    connection.connect();
```

Once the connection has been established, the response can simply be copied from the connection's input stream:

```
    Reader responseReader = new
InputStreamReader( connection.getInputStream() );
    StringWriter stringWriter = new StringWriter();
    char[] buffer = new char[4096];
    int n = 0;
    while( -1 != ( n = responseReader.read( buffer ) ) )
    {
          stringWriter.write( buffer, 0, n );
    }
    String response = stringWriter.toString();
```

This convenience method prints the response code and associated message before returning the response to the caller:

```
System.out.println( "----------------------------------------" );
System.out.println( connection.getResponseCode() + " " +
connection.getResponseMessage() );
System.out.println( "----------------------------------------" );
connection.disconnect();
```

This would normally be an HTTP code of 200 / OK:

```
----------------------------------------
200 OK
----------------------------------------
```

In a real application, receiving the response through an HTTP request is very easy. The client code goes a step further by demonstrating how the response can be parsed into known object types for use in Java. When Retrieving Gas Stations through REST by using the standalone client, the first argument is *stations* and is used by the *main* method of the client class to handle the request:

```
...
else if( args[0].equals( "stations" ) )
{
    String latitude = args[1];
    String longitude = args[2];
    Coordinates coordinates = new Coordinates( latitude, longitude );
    String username = args[3];
    String password = args[4];
    getGasStationsREST( coordinates, username, password );
}
```

The second and third arguments have to be the latitude and longitude of the grid cell center for which gas stations are being queried. The fourth and fifth arguments are the user ID and password of a user with the *REST* security role.

The query method reads the bundled property file for default and expects a Java system property to override the JDG host (and/or port) as appropriate:

```
private static void getGasStationsREST(Coordinates coordinates, String
username, String password) throws IOException
{
    Properties properties = new Properties();
    properties.load( Client.class.getResourceAsStream(
                                                  "/jdg.properties" ) );
    String host = properties.getProperty( "jdg.host" );
    host = System.getProperty( "jdg.host", host );
```

The address for the REST request is then formed based on these values:

```
String urlServerAddress =
    "http://" + host + ":8080/rest/stations/" + coordinates.asString();
```

At this point, a simple call to the convenience method returns the response in JSON format:

```
String response = restGet( urlServerAddress, username, password );
```

The response is a JSON map. The **gson** library is used to first parse the response as a generic JSON object and then iterate through each entry. Each entry is then parsed into a *GasStation* object, which is in turned stored in a map of gas stations:

```
Map<UUID, GasStation> gasStations = new HashMap<UUID, GasStation>();
JsonObject jsonObject = new JsonParser().parse( response )
                                                .getAsJsonObject();
for( Entry<String, JsonElement> jsonElementEntry : jsonObject.entrySet() )
{
    JsonElement jsonElement = jsonElementEntry.getValue();
    GasStation gasStation = (GasStation)new Gson()
                                .fromJson( jsonElement, GasStation.class );
    gasStations.put( gasStation.getStationId(), gasStation );
}
```

The method proceeds to print aggregate information and a sample of the results:

```
System.out.println( "REST response parsed to " + gasStations.size() + " gas
stations, here is a random example:" );
System.out.println( gasStations.values().iterator().next() );
System.out.println( "\n\nAll station UUIDs:\n*************************" );
for( GasStation gasStation : gasStations.values() )
{
    System.out.println( gasStation.getStationId() );
}
System.out.println( "*************************\n" );
```

For each gas station, its ID is printed. These unique identifier are the cache key to the most recent price report for their respective gas station.

When Retrieving Price Reports through REST by using the standalone client, the first argument is *prices* and is used by the *main* method of the client class to handle the request:

```
...
else if( args[0].equals( "prices" ) )
{
     String uuid = args[1];
     String username = args[2];
     String password = args[3];
     getPriceReportREST( uuid, username, password );
}
```

The second argument is the cache key, which is the UUID of the gas station when querying the most recent price report and the same value with an underscore and the timestamp appended for earlier price reports. The third and fourth arguments are the user ID and password of a user with the *REST* security role.

The query method reads the bundled property file for default and expects a Java system property to override the JDG host (and/or port) as appropriate:

```
private static void getPriceReportREST(String uuid, String username, String
password) throws IOException
{
     Properties properties = new Properties();
     properties.load( Client.class.getResourceAsStream(
                                                  "/jdg.properties" ) );
     String host = properties.getProperty( "jdg.host" );
     host = System.getProperty( "jdg.host", host );
```

The address for the REST request is then formed based on these values:

```
String urlServerAddress = "http://" + host + ":8080/rest/prices/" + uuid;
```

At this point, a simple call to the convenience method returns the response in JSON format:

```
String response = restGet( urlServerAddress, username, password );
System.out.println( response );
```

The response is a single JSON object that is printed on the console. The **gson** library can be used to parse the JSON into a *PriceReport* object but the JSON representation of the object is slightly different in gson as opposed to JDG. While JDG represents a Java date object by the number of millisecond passed since January 1, 1970, gson expects a (configurable) text representation of the date. The easiest solution is to override the behavior and parse the date field through custom code. To do this, create a gson date deserializer:

```
JsonDeserializer<Date> dateDeserializer = new JsonDeserializer<Date>()
{
...
```

The implementation simply takes the milliseconds value and creates and returns a Java Date object:

```
    @Override
    public Date deserialize(JsonElement jsonElement, Type type,
JsonDeserializationContext context) throws JsonParseException
    {
          return new Date( jsonElement.getAsLong() );
    }
};
```

Before using gson to parse the response, configure it to use the above deserializer:

```
    GsonBuilder gsonBuilder = new GsonBuilder();
    gsonBuilder.registerTypeAdapter( Date.class, dateDeserializer );
    Gson gson = gsonBuilder.create();
```

Once this is configured, it is very easy to parse the JSON into the desired object type:

```
PriceReport priceReport =
               (PriceReport)gson.fromJson( response, PriceReport.class );
```

The result is printed to the console:

```
    System.out.println( "\n\nREST response parsed to PriceReport object:" );
    System.out.println( "**************************" );
    System.out.println( priceReport );
    System.out.println( "**************************" );
```

5.5 Cache Size Query

A distributed cache copies each entry to as many cluster nodes as configured for the number of owners. If the size of the cluster exceeds the number of owners configured for the cache distribution, no JDG node would hold all the cache entries locally. The process of finding and retrieving an entry is transparent to the caller, however some of the distribution is exposed in the response to a cache size query.

The standalone client code prints the cache size information when it receives the *size* argument:

```
public static void main(String[] args) throws Exception
{
     Client loader = new Client();
     if( args.length > 0 )
     {
           else if( args[0].equals( "size" ) )
           {
                 ...
```

Both *stations* and *prices* caches are referenced:

```
String[] caches = new String[] {"stations", "prices"};
```

For each cache, the available convenience method in the class is used to obtain a *RemoteCache* reference:

```
for( String cache : caches )
{
     RemoteCache<String, Object> remoteCache = getCache( cache );
```

This method creates the cache reference by connecting to either localhost or the JDG server running on the address specified by the *jdg.host* system property. Calling the size() method on the remote cache only returns the number of entries owned and located by this single cluster node:

```
     int nodeSize = remoteCache.size();
```

To obtain the total cache size, the set of cache keys are requested and counted:

```
     int totalSize = remoteCache.keySet().size();
```

The results are printed to the console:

```
     System.out.println( cache + " cache size is " + totalSize + " and "
      + nodeSize + " items are stored directly on the given JDG node" );
}
```

6 Other Technical Notes

6.1 Horizontal Scaling

An important benefit of running JBoss Data Grid in remote client-server mode is the ability to scale out the JDG cluster with no impact to the business application.

This reference architecture assumes that the size of the *prices* cache may grow exponentially in the future and takes advantage of the scalability of JDG to maintain consistent and constant response times.

In Distribution Mode, the total amount of memory occupied by cache data in a cluster of JBoss Data Grid servers is fixed at the size of data multiple by the number of owners configured for the cache. As a result, an increase of a given percentage in the size of cache data can simply be handled by increasing the number of nodes by the same percentage without accounting for any cluster overhead.

The Hot Rod client remains aware of the cluster topology and maintains the same consistent hashing algorithm. This removes the overhead associated with the distribution of data as opposed to its replication; the client can calculate where the target data is located and contact a JDG node that owns the data directly, without incurring the cost of any extra roundtrips.

6.2 Initial State Transfer

State transfer occurs automatically in Red Hat JBoss Data Grid whenever a node joins or leaves the cluster.[35]

In distribution mode, the addition or removal of a node from the cluster changes the cluster size and depending on the number of owners configured for the cache, prompts a redistribution of data. When a new node is added, some of the load is moved from the previously existing nodes to the new node. Once a node leaves the cluster, its responsibility as one of the owners of some cache entries is transferred to another node.

In replication mode, the removal of a node does not cause any state transfer but the entire cache content is copied and transferred to a node that joins the cluster.

Start only two nodes of the reference architecture environment cluster and load sample data as outlined in Sample Data Population.

At this point, the cluster size and the number of owners for the price cache are both set to two and querying the Cache Size confirms that both nodes hold the full cache:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar size
```

Running this on either of the two nodes results in the same response:

```
...
prices cache size is 179180 and 179180 items are stored directly on the
given JDG node
```

35 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-State_Transfer.html

Start the third node after confirming the cache size with only two nodes. As the third node joins the cluster, state transfer is initiated and the data in the *prices* cache is redistributed. The number of owners is set to 2 and there will be 3 nodes in the cluster, so each entry will be available on 2 of the 3 nodes and it follows that each node will contain roughly two-thirds the number of cache entries. Once again, query the cache size:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar size
```

Notice the approximate ratio of the number of entries on the node to the total:

```
...
prices cache size is 179180 and 116534 items are stored directly on the
given JDG node
```

As expected, 116534 is roughly two-thirds of 179180.

Now stop another node of the cluster. Assuming that nodes 1 and 2 were initially started and node 3 later joined the cluster, try killing the process for node 2 and notice the state transfer that follows:

```
# java -Djdg.host=10.19.139.101 -jar GasShopperClient.jar size
```

Running this on either of the two nodes results in the same response:

```
...
prices cache size is 179180 and 179180 items are stored directly on the
given JDG node
```

With the cluster size now being the same as the number of owners configured for the *prices* cache, all data is once again stored on each node.

6.2.1 Suppressing State Transfer

While state transfer is a useful and necessary feature of JBoss Data Grid, it is also a time consuming and resource intensive operation. When servers are intentionally shut down, state transfer can result in a very lengthy graceful shutdown process and eventually cause a number of memory errors on the surviving nodes. When the data stored in a distributed cache is too large for a single node, as will typically be the case in a large deployment, shutting down server and reducing the cluster size inevitably causes a scenario where the remaining nodes can no longer handle the load and are flooded with out of memory exceptions.

To avoid state transfer when a node joins or leaves the cluster, dynamically set the rebalancingEnabled JMX attribute of the LocalTopologyManager MBean to false.[36] In JBoss Data Grid, JMX is used in conjunction with JBoss Operations Network (JON) to expose this information and present it in an orderly and relevant manner to the administrator.[37]

36 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/sect-State_Transfer.html#The_rebalancingEnabled_Attribute

37 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.3/html/Administration_and_Configuration_Guide/Using_JMX_with_JBoss_Data_Grid1.html

6.3 Cross-Datacenter Replication

It is common for multiple copies of sensitive data to be stored in geographically dispersed locations in order to survive a catastrophic events.

In JBoss Data Grid, Cross-Datacenter Replication allows the administrator to create data backups in multiple clusters. These clusters can be at the same physical location or different ones.

Configure each location through the **RELAY** protocol of **JGroups**:

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.2"
        default-stack="udp">
    <stack name="udp">
          <transport type="UDP"
                   socket-binding="jgroups-udp"/>
          ...
          <relay site="LON">
                <remote-site name="NYC"
                          stack="tcp"
                        cluster="global"/>
                <remote-site name="SFO"
                          stack="tcp"
                        cluster="global"/>
          </relay>
    </stack>
</subsystem>
```

This protocol uses TCP to connect the local group to a remote site, as configured.

On each site, configure the distributed cache and specify the backup sites that are used:

```
<distributed-cache>
     ...
     <backups>
        <backup site="{FIRSTSITENAME}" strategy="{SYNC/ASYNC}" />
        <backup site="{SECONDSITENAME}" strategy="{SYNC/ASYNC}" />
     </backups>
</distributed-cache>
```

The local site transport is configured separately:

```
<transport executor="infinispan-transport"
           lock-timeout="60000"
           cluster="LON"
           stack="udp"/>
```

Refer to the official Red Hat documentation for further details on configuring and using corss-datacenter replication for JBoss Data Grid.[38]

38 https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Data_Grid/6.1/html/Administration_and_Configuration_Guide/sect-Configure_Cross_Datacentre_Replication.html

6.4 Data Security

In its default configuration, JBoss Data Grid server binds both **Hot Rod** and **memcached** protocols to the management interface:

```
<socket-binding-group name="standard-sockets" default-interface="public"
                      port-offset="${jboss.socket.binding.port-offset:0}">
...
...
    <socket-binding name="hotrod" interface="management" port="11222"/>
...
    <socket-binding name="memcached" interface="management" port="11211"/>
```

The management interface is configured separately from the public interface and while the public interface is typically bound to the network or external IP address of the machine while starting the server, the management interface may remain unchanged:

```
<interfaces>
    <interface name="management">
        <inet-address value="${jboss.bind.address.management:127.0.0.1}"/>
    </interface>
    <interface name="public">
        <inet-address value="${jboss.bind.address:127.0.0.1}"/>
    </interface>
</interfaces>
```

This reference environment changes the *hotrod* socket binding interface to *public* so that it may be accessed from the private network to which it is bound.

The strategy for REST access to cache data is different as access is provided through the HTTP protocol which is bound to the public interface by default:

```
<socket-binding name="http" port="8080"/>
```

Interacting with the cache through REST over HTTP is secured by default:

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.0">
    <hotrod-connector socket-binding="hotrod" cache-container="clustered">
        <topology-state-transfer lazy-retrieval="false" lock-timeout="1000"
                                 replication-timeout="5000"/>
    </hotrod-connector>
    <memcached-connector socket-binding="memcached"
                                              cache-container="clustered"/>
    <rest-connector virtual-server="default-host"
                            cache-container="clustered"
                            security-domain="other" auth-method="BASIC"/>
</subsystem>
```

This reference environment creates an application user with the security role of *REST* to access data through this method and pass authorization.

JBoss Data Grid 6.2 provides security through TLS/SSL client certificate-based authentication and authorization. Data traffic between the JDG client and the server may also be encrypted by TLS/SSL to provide confidentiality. JBoss Data Grid 6.3 enhances security even further with several new features:

6.4.1 Role-based access control

Red Hat JBoss Data Grid 6.3 features role-based access control for operations on designated secured caches. This feature allows custom security roles to be defined and associated individually with cache and cache-manager operations.

The JBoss Data Grid server can be configured to use file-based, database and ldap security stores or a combination of them to authenticate users and define and obtain their roles. The derived role is then matched against the configured constraints before allowing a secured cache manager or cache operation.

6.4.2 Node Authentication and Authorization

Red Hat JBoss Data Grid 6.3 supports cluster node authentication and authorization.

To prevent backdoor security breaches, the cluster can be configured to require authentication and authorization before a node is allowed to join and receive data.

6.4.3 Encrypted Communication Within the Cluster

Red Hat JBoss Data Grid 6.3 further increases data security by providing support for encrypted data transfer between cluster nodes.

Data encryption between cluster nodes may be customized by users within the limits of support provided by the **`Java Cryptography Architecture (JCA)`**.

6.5 Compatibility Mode

By default, REST, Memcached, and Hot Rod each store data in the most efficient format for that protocol, ensuring transformations are not required when retrieving entries. When this data is required to be accessed from multiple protocols, compatibility mode must be enabled on caches that are being shared.

Compatibility mode comes at a higher performance cost than non-compatibility mode. As a result, this feature is disabled by default in JBoss Data Grid.

To enable compatibility mode for a given cache, simply add the following configuration:

```
<distributed-cache name="prices" ...>
    <compatibility enabled="true"/>
```

This reference environment enables compatibility mode for both the *stations* and *prices* cache so that the data, stored through Hot Rod, may be queried using REST over HTTP.

7 Conclusion

This reference environment sets up a cluster of JDG servers running for remote client-server use and configures both distributed and replicated caches on the servers. The reference architecture defines a business use case and builds a complete application around it with a focus on leveraging JBoss Data Grid.

Real life usage of any product involves a large variety of considerations and this reference architecture attempts to address many of them for the given use case by walking through every step of the cache set up and interaction, including designing the data model, configuring the server to support custom cache values, leveraging various patterns to balance efficiency and ease of use in cache communication and solving any challenges encounted in the process.

Appendix A: Revision History

Revision 1.0	07/25/14	Babak Mozaffari
Initial Release		

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
Divya Mehra	Principal Product Manager – Technical	Review
Mircea Markus	Principal Software Engineer	Technical Review
Tristan Tarrant	Principal Software Engineer	Technical Review
Martin Gencur	Senior Quality Engineer	Technical Review

Appendix C: IPTables configuration

An ideal firewall configuration constraints open ports to the required services based on respective clients. This reference environment includes a set of ports opened for the JDG cluster to allow clients access through various protocols, including Hot Rod and REST / HTTP. Other than the TCP ports accessed by callers, there are also a number of UDP ports that are used within the cluster itself for replication, failure detection and other HA functions. The following iptables configuration opens these ports to the other servers on the same subnet.

```
# Generated by iptables-save v1.4.7 on Wed Jun 18 22:54:42 2014
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [138:27424]
-A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -p tcp -m state --state NEW -m tcp --dport 22 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 8080 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 11222 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 54200 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 4447 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 4712 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p tcp -m tcp --dport 4713 -j ACCEPT
-A INPUT -s 234.99.54.14 -p udp -m udp --dport 45688 -j ACCEPT
-A INPUT -s 10.19.139.0/24 -p udp -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
# Completed on Wed Jun 18 22:54:42 2014
```

Note: For a cluster of nodes running on **`Linux`** virtual machines, certain kernel and **`oVirt`** or **`KVM`** versions may manifest UDP multicast issues where the UDP traffic flow abruptly stops after 4-5 minutes. The practical result is that the nodes stop seeing each other and the cluster breaks up. Similar issues are reported in Bugzilla case #880035[39] and it has been observed in a virtualized environment in **`RHEL 6.5`**.

To work around this issue, disable IGMP Snooping by finding the bridge that is used by the guest VMs and running the following command on the host:

```
# brctl setmcsnoop virbr0 0
```

Replace *virbr0* with the correct bridge interface.

39 https://bugzilla.redhat.com/show_bug.cgi?id=880035