CONTENTS

Infinispan

BY MANIK SURTANI, UPDATED AND REVISED BY TRISTAN TARRANT

WHAT IS INFINISPAN

Infinispan is an open-source, ASL 2.0-licensed, in-memory data grid platform. In-memory data grids are commonly used as low-latency, highly available and elastic data storage backends, often as NoSQL solutions. A common use is in addition to traditional databases, as a distributed cache for fast data access.

Infinispan can be used either embedded in your application, or remotely, via an efficient high-performance protocol known as Hot Rod, or over more ubiquitous protocols like Memcached. Your choice on which mode of interaction to use will depend on a number of factors, including whether you are using Infinispan as a clustering toolkit to cluster your own framework, whether you intend to use Infinispan to cache database lookups, or whether you plan to interact with Infinispan from a non-JVM environment. You can interact with Infinispan directly through its rich API, or use one of several integrations such as JCache, Spring Cache, and CDI.

HOW CAN I GET IT?

Depending on how you intend to use Infinispan, there are different ways you can download it. If you are embedding Infinispan in your application, the most convenient way is to use Maven.

Infinispan is available from Maven Central at the following coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-core</artifactId>
  <version>LATEST_INFINISPAN_VERSION</version>
</dependency>
```

If instead you prefer to manage your project dependencies by yourself, or you want to use Infinispan Server, head over to infinispan.org/download to download your preferred distribution.

EMBEDDED MODE

When used in this mode, Infinispan instances reside within your JVM alongside your code. You start up multiple instances of your application on different servers and each one starts and initializes an Infinispan instance. These Infinispan instances discover each other, form a data grid, and start sharing and distributing data. This mode is what you will want to use if you want an in-memory cache (either local or clustered) to front a database or some other expensive data source or if you are building an application or a framework that needs to be cluster-aware.

A PRACTICAL EXAMPLE

To see this in action, make sure the Infinispan libraries are available to your code—either via Maven, as above, or via downloading the zipped distribution and extracting the JAR files.

Starting Infinispan instances from your code is simple. To start a local, non-clustered cache instance:

```
DefaultCacheManager m = new DefaultCacheManager();
Cache<String, String> c = m.getCache();
c.put("hello", "world");
```

Infinispan's core construct is an instance of the Cache interface. Extending `java.util.concurrent.ConcurrentMap`, Cache exposes well-known methods to store and retrieve objects.

To start a cluster-aware instance capable of detecting neighboring instances on the same local area network and sharing state between them:

```
GlobalConfigurationBuilder global =
GlobalConfigurationBuilder.getClusteredDefault();
ConfigurationBuilder config = new ConfigurationBuilder();
config.clustering().cacheMode(CacheMode.DIST_SYNC);
DefaultCacheManager m = new DefaultCacheManager(
  global.build(), config.build());
Cache<String, String> cache = m.getCache();
cache.put("hello", "world");
cache.containsKey("hello"); // returns true
cache.get("hello"); // returns "world"
cache.remove("hello"); // returns "world"
```

This can also be done using a configuration file on the classpath:

```
DefaultCacheManager m = new DefaultCacheManager(
  "path/to/my/infinispan_config.xml");
Cache<String, String> c = m.getCache();
```

Where the configuration file "infinispan_config.xml" contains:

```
<infinispan>
  <cache-container default-cache="default">
  <transport cluster="mycluster"/>
  <distributed-cache name="default" mode="SYNC"/>
</cache-container>
</infinispan>
```

Important: Infinispan's clustering is implemented on top of the powerful JGroups library, which supports a large number of discovery protocols, depending on your environment. Check it out at jgroups.org.

KEY APIS

`org.infinispan.Cache<?,?>`	The main cache interface
`org.infinispan.manager.DefaultCacheManager`	The cache manager implementation. Responsible for managing the configuration and lifecycle of all aspects of Infinispan (caches, clustering, etc.)
`org.infinispan.configuration.cache.ConfigurationBuilder`	The cache configuration builder, used to programmatically define the features of each cache.
`org.infinispan.configuration.global.GlobalConfigurationBuilder`	The cache manager configuration builder, used to programmatically define global features such as thread pools, clustering, etc.
`@org.infinispan.notifications.Listener`	The annotation to put on event listeners
`org.infinispan.notifications.cachelistener.event.*`	All interfaces in this package define a type of Cache event which can be processed by a custom event listener
`org.infinispan.notifications.cachemanagerlistener.event.*`	All interfaces in this package define a type of CacheManager event which can be processed by a custom event listener
`org.infinispan.commons.marshall.Marshaller`	The interface that defines a marshaller (serializer/deserializer) for custom data put in a cache
`org.infinispan.query.SearchManager`	The entry point to create queries on top of a cache

CACHE FEATURES

Infinispan caches are very powerful and can be configured and tuned to suit the needs of most applications. Let's look at some of the features you can enable:

EXPIRATION

Usually the purpose of a cache is to store short-lived data that needs to be refreshed regularly. You can tell Infinispan to store an entry only for a certain period of time:

```
cache.put("mortal", "human", 10, TimeUnit.MINUTES);
```

You can also set a default expiration lifespan for all entries in a cache via the configuration:

```
configurationBuilder.expiration()
  .lifespan(10, TimeUnit.MINUTES);
```

or in XML:

```
<distributed-cache name="mortaldata">
  <expiration lifespan="600000" />
  <!-- lifespan in milliseconds -->
</distributed-cache>
```

EVICTION

Normally, caches are unbounded, i.e. they grow indefinitely and it is up to the application to remove unneeded data. In Infinispan you can also set a maximum size for a cache: when this limit is reached, entries accessed least recently will be evicted. You can set the size based on number of entries:

```
configurationBuilder.eviction()
  .strategy(EvictionStrategy.LRU)
  .type(EvictionType.COUNT)
  .size(1000);
```

or in XML:

```
<distributed-cache name="boundedcache">
  <eviction strategy="LIRS" type="COUNT" size="1000" />
  <!-- maximum 1000 entries -->
</distributed-cache>
```

You can also set a maximum memory occupation. This currently only works for some "well known" data-types (`String`, `byte[]`) or when the *storeAsBinary* configuration option is enabled:

```
configurationBuilder.eviction()
  .strategy(EvictionStrategy.LIRS)
  .type(EvictionType.MEMORY)
  .size(1000000);
```

or in XML:

```
<distributed-cache name="boundedcache">
  <eviction strategy="LIRS" type="MEMORY" size="1000000" />
  <!-- maximum 1MB -->
</distributed-cache>
```

PERSISTENCE

A cache, being an in-memory data structure, will lose data when stopped. You can overcome this limitation by backing a cache with a persistent store. Depending on configuration the store will act either as a persistent copy of your in-memory data or, in combination with eviction, as an overflow for excess data that you don't want to keep in memory. There is a multitude of available cache stores available, depending on the features you need, ranging from file-based, to JDBC, to popular NoSQL and Cloud stores like MongoDB, S3, and Cassandra. Here are some examples:

Simple write-through file-based store:

```
configurationBuilder.persistence().addSingleFileStore()
  .location("/path/to/persistent/data");
```

or in XML:

```
<distributed-cache name="filecache">
  <persistence>
    <file-store path="/path/to/persistent/data" />
  </persistence>
</distributed-cache>
```

Write-behind JDBC store with keys represented as strings using the internal connection pool:

```
configurationBuilder.persistence()
  .addStore(JdbcStringBasedStoreConfigurationBuilder.class)
  .connectionPool()
  .connectionUrl("jdbc:postgresql://dbhost:5432/infinispan")
  .username("infinispan")
  .password("infinispan")
  .driverClass(org.postgresql.Driver.class)
  .table().tableNamePrefix("ISPN_STRING_TABLE")
  .createOnStart(true)
  .async().enabled();
```

or in XML:

```
<distributed-cache name="filecache">
  <persistence>
    <string-keyed-jdbc-store>
      <connection-pool
        connection-url=
          "jdbc:postgresql://dbhost:5432/infinispan"
        username="infinispan"
        password="infinispan"
        driver="org.postgresql.Driver"/>
      <string-keyed-table create-on-start="true"
        prefix="ISPN_STRING_TABLE" />
      <write-behind />
    </string-keyed-jdbc-store>
  </persistence>
</distributed-cache>
```

TRANSACTIONS

Infinispan can be configured to use and participate in JTA transactions. This means that you can perform multiple operations on a cache (or multiple caches) and commit them (or roll them back) atomically. You can either use the internal (but limited) DummyTransactionManager or an external transaction manager, like the one provided by your container. Configure a transactional cache with:

```
configurationBuilder.transaction()
  .transactionMode(TransactionMode.TRANSACTIONAL);
```

or in XML:

```
<distributed-cache name="transactional">
  <transaction mode="NON_XA" />
</distributed-cache>
```

and then you interact with a cache transactionally as follows:

```
TransactionManager tm = cache.getAdvancedCache().
getTransactionManager();
tm.begin();
int i = cache.get("k1");
cache.put("k1", i++);
tm.commit();
```

QUERIES AND INDEXING

Infinispan supports searching of Java objects stored in the grid using powerful search APIs which complement its main Cache API. Supposing your cache contains *Person* objects:

```
public class Person {
  String name;
  String surname;
  Date birthday;
}
```

you can query it by obtaining a *QueryFactory*:

```
QueryFactory qf = org.infinispan.query
  .Search.getQueryFactory(cache);
```

then use the query DSL to build the query which will retrieve all *Person* objects whose name starts with *J* born between 1970 and 1975:

```
org.infinispan.query.dsl.Query query = qf.from(Person.class)
  .having("name").like("J%")
  .and().having("birthday")
  .between(Year.of(1970), Year.of(1975))
  .toBuilder().build();
List<Person> list = query.list();
```

By default, queries will apply the predicates to all of the entries to find matches. You can also enable the use of Lucene indexes which will provide a significant performance boost along with a bunch of additional features.

First, enable indexing as follows:

```
configurationBuilder.indexing().enable()
  .index(Index.LOCAL)
  .autoConfig(true);
```

or in XML:

```
<distributed-cache name="indexedcache">
  <indexing index="LOCAL" auto-config="true" />
</distributed-cache>
```

then annotate the entities you are going to store in the cache by specifying which fields to index:

```
@Indexed
public class Person {
  @Field String name;
  @Field String surname;
  @Field Date birthday;
}
```

EVENTS AND LISTENERS

Infinispan offers a listener API, where clients can register for and get notified when events such as entries being modified or nodes joining/leaving the cluster take place. You don't need to configure anything in order to be able to register events on a cache. For example, the following class defines a listener to print out some information every time a new entry is added to the cache:

```
@Listener
public class PrintWhenAdded {
  @CacheEntryCreated
  public void print(CacheEntryCreatedEvent event) {
    if (!event.isPre())
    System.out.println("New entry "
      + event.getKey()
      + " created in the cache");
  }
}
```

And you register the above listener on a cache with:

```
cache.addListener(new PrintWhenAdded());
```

Listener annotations have some attributes which affect their behavior. Most events will fire both before and after the actual operation is performed. You can tune this behavior by specifying an observation attribute:

```
@Listener(observation = Observation.POST)
```

Normally, listeners are local, i.e. they will receive events for affected entries only if they are owned by the node where the listener has

been registered. You can register a listener that will listen to events happening on the entire cluster by using:

```
@Listener(clustered = true)
```

but be aware that this will increase cluster traffic considerably. Note that cluster listeners only support POST observation.

EXECUTORS

You can execute arbitrary code across your cluster using the ClusterExecutor. Just submit a *java.util.function.Function*, which receives the local CacheManager as input and can return any value to the invoker:

```
manager.executor().submitConsumer(localManager -> {
  // Perform some meaningful computation local to each node
  String s = ... ;
  return s;
}, (address, value, throwable) -> {
  if (throwable != null) {
    log.fatal("Encountered error while processing on node "
    + address, t);
  } else {
    log.info("Node %s has returned %s", address, value);
  }
}).join();
```

The code you run doesn't necessarily need to interact with the caches themselves.

CACHING MODES

Infinispan offers four caching modes: local, invalidation, replicated, and distributed. All of the modes support all of the features described above.

LOCAL CACHES

As the name implies, these are entirely local, i.e. they do not share data with any other node, even when the CacheManager is being clustered. Because of this, they are also the fastest type of cache. A local cache is configured as follows:

```
cacheManager.defineConfiguration("localcache",
  new ConfigurationBuilder().build());
Cache<String, String> cache =
  cacheManager.getCache("localcache");
```

or in XML:

```
<local-cache name="localcache" />
```

Local caches can use all of the features of Infinispan (persistence, transactions, indexing, etc.). If, however, you don't need any of those, you can use "simple caches" to squeeze even more performance. Simple caches support listeners, expiration, eviction, and security. Turn them on with:

```
cacheManager.defineConfiguration("simplecache",
  new ConfigurationBuilder().simpleCache(true).build());
Cache<String, String> cache =
  cacheManager.getCache("simplecache");
```

or in XML:

```
<local-cache name="simplecache" simple-cache="true" />
```

INVALIDATION CACHES

An invalidation cache is a special type of clustered cache that does not share any data: only invalidation messages are sent to other members in the cluster when a node modifies/removes an entry, thus avoiding stale data. An invalidation cache is configured as follows:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.clustering().cacheMode(CacheMode.INVALIDATION_SYNC);
cacheManager.defineConfiguration("invcache", cfg.build());
Cache<String, String> cache =
  cacheManager.getCache("invcache");
```

or in XML:

```
<invalidation-cache name="invcache" mode="SYNC"/>
```

REPLICATED CACHES

In replicated caches, entries added on any node are replicated to all other nodes in the cluster. This ensures that reads will always be performed locally. Use this type of cache for read-heavy data (e.g. lookup tables) where writes are infrequent and when you don't need to scale capacity horizontally. A replicated cache is configured as follows:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.clustering().cacheMode(CacheMode.REPL_SYNC);
cacheManager.defineConfiguration("replcache", cfg.build());
Cache<String, String> cache =
  cacheManager.getCache("replcache");
```

or in XML:

```
<replicated-cache name="replcache" mode="SYNC" />
```

DISTRIBUTED CACHES

Distributed caches allow you to scale capacity linearly by adding new nodes to the cluster. You only need to specify the number of owners (i.e. copies of data) you require. Infinispan will perform automatic rebalancing of data when nodes are added/removed to ensure that availability isn't compromised. A distributed cache is configured as follows:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.clustering()
  .cacheMode(CacheMode.DIST_SYNC).hash().numOwners(2);
cacheManager.defineConfiguration("distcache", cfg.build());
Cache<String, String> cache =
  cacheManager.getCache("distcache");
```

or in XML:

```
<distributed-cache name="distcache"
  mode="SYNC" num-owners="2" />
```

SYNCHRONOUS AND ASYNCHRONOUS CLUSTERING

For each of the clustered modes (invalidation, replicated, and distributed) you may choose whether you want communication between the nodes to be performed synchronously or asynchronously. Synchronous mode is slower, but safer, since write operations wait for completion on the remote nodes before returning. If you want more performance and can afford potentially losing writes on other nodes in case of failures, use asynchronous mode.

WHICH MODE SHOULD I USE?

The following table summarizes the characteristics of each cache mode, so that you can choose the one that more closely meets your application's needs:

MODE	CLUSTERED	MAXIMUM CAPACITY	AVAILABILITY	PERFORMANCE
Local	No	Limited by the the individual node's memory	Lowest, as there is only one copy of the data	Maximum in reads and writes
Invalidation	Yes, but without sharing data	Limited by each node's individual memory	Low	Fast reads, writes require remote calls
Replicated	Yes, all nodes have the same data	Limited by the node with the least memory	Maximum, can survive failures of all nodes but one	Fast reads, slowest writes
Distributed	Yes, data is distributed in the cluster	Limited only by the number of available nodes	High, can survive concurrent failures of owners-1 nodes	Reads and writes might require remote calls

CACHES AND THE STREAMS API

Since Infinispan 8 is based on Java 8, you can use the very powerful Java Streams API to perform mass filtering and computations on a Cache, by taking advantage of data distribution and locality. For example, the following piece of code:

```
cache.entrySet().stream()
  .map(e -> e.getValue())
  // we only need the value
  .map(v -> v.toLowerCase())
  // convert it to lowercase
  .map(v -> v.split("s+"))
  // split each value in single words
  .flatMap(a -> Arrays.stream(a))
  // create a new stream from the array of single words
  .collect(Collectors.groupingBy(
    Function.identity(), Collectors.counting()));
  // count equal words
```

performs the usual "word count" on all values of a cache. On a distributed Infinispan cache this will run in parallel on all nodes, using all the computing power of your cluster.

JCACHE (JSR-107)

Aside from the very powerful Cache API, you may also wish to use the standard JCache API, which offers an implementation-independent way to create, retrieve and manipulate caches. Ensure that your application depends on the infinispan-jcache JAR and use it as follows:

```
CachingProvider jcacheProvider =
  Caching.getCachingProvider();
CacheManager cacheManager = jcacheProvider.getCacheManager();
MutableConfiguration<String, String> configuration =
  new MutableConfiguration<>();
configuration.setTypes(String.class, String.class);
Cache<String, String> cache =
  cacheManager.createCache("myCache", configuration);
cache.put("key", "value");
System.out.printf("key = %sn", cache.get("key"));
cacheManager.close();
```

CLIENT/SERVER MODE

You may not always want Infinispan instances to reside in the same JVM as your application. Sometimes this is for security, sometimes for architectural reasons to maintain a separate data layer, but this can also be because your client application is not on a JVM platform. For example, Node.js, C++, or .NET clients can also make use of Infinispan if it is run as a remote server.

Infinispan Server comes with several different server endpoints, speaking a variety of protocols. Here is a comparison of the protocols that can be used:

PROTOCOL	FORMAT	CLIENTS	CLUSTERED	SMART ROUTING	BALANCING/ FAILOVER
REST	Text	Lots	Yes	No	Any HTTP balancer
Mem-cached	Text	Lots	Yes	No	Only with predefined list of servers
Hot Rod	Binary	Java, C++, .NET, JavaScript	Yes	Yes	Dynamic, via Hot Rod clients
WebSocket	Text	JavaScript	Yes	No	Any HTTP balancer

Infinispan Server is built on the robust foundation of WildFly, so it comes out of the box complete with management, monitoring and security features. To start a two-node cluster on your machine simply unzip the distribution and run the "domain" script:

```
$ bin/domain.sh
```

Infinispan Server comes with an easy to use administration console. Before using it you need to create an admin user:

```
$ bin/add-user.sh -u admin -p secret
```

At this point you can connect to the console by using the URL http://localhost:9990 in your browser.

Using Infinispan over the Hot Rod protocol is similar to using the embedded API. First of all, you need to depend on the client JAR:

```
<dependencies>
  …
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-client-hotrod</artifactId>
    <version>LATEST_INFINISPAN_VERSION</version>
  </dependency>
  …
</dependencies>
```

And then use it in your application as follows:

```
RemoteCacheManager rcm =
  new RemoteCacheManager("infinispan_host");
RemoteCache rc = rcm.getCache();
rc.put("hello", "world");
String value = rc.get("hello");
```

The Hot Rod protocol has been designed specifically with load balancing and failover in mind. Clients can be written to take advantage of the server topology that is provided to clients and regularly kept up-to-date. Clients can even be made aware of hash functions used on the backend, so routing requests to a cluster of backend nodes can be done in an intelligent fashion to minimize latency and remote lookup on the backend. All of the available clients (Java, C++, .NET, and Node.js) make use of such features and provide built-in load-balancing, failover, discovery of new backend nodes, as well as intelligent routing of requests.

If you need to share data among different languages/platforms, the use of a common marshalling format is recommended. While Infinispan does not impose a default, one option is ProtoBuf, a compact format with wide support.

CONTAINERS AND CLOUD ENVIRONMENTS

If you like running your applications in containers, Infinispan comes with a well-maintained container image that makes it easy to run Infinispan Servers in clustered, domain, and standalone modes, with different protocol stacks.

By default, the container runs in clustered mode, and to start a node simply execute:

```
docker run -it jboss/infinispan-server
```

Starting a second container will cause it to form a cluster with the first one. You can check the membership status of the cluster by running a command directly in the newly launched container:

```
docker exec -it $(docker ps -q -l) /opt/jboss/infinispan-
server/bin/ispn-cli.sh -c "/subsystem=datagrid-infinispan/
cache-container=clustered:read-attribute(name=members)"
```

Which should output something like this:

```
{
  "outcome" => "success",
  "result" => "[57b9a116303f, 7648d4b62987]"
}
```

The above examples will use UDP Multicast to perform discovery and cluster communication. If, however, you're running in a cloud environment or another environment that does not allow the use of multicast, you can use the JGroups Gossip Router as an alternative discovery mechanism. Employing a gossip router will enable discovery via TCP, where the router acts as a registry: each member will register itself in this registry upon start and also discover other members.

Launch the gossip router container as follows:

```
docker run -ti -p 12001:12001 jboss/jgroups-gossip:3.6.10.
Final
```

Take note of the address where the router will bind to, it's needed by the Infinispan nodes. The address can be easily obtained by:

```
docker inspect --format '{{ .NetworkSettings.IPAddress }}'
$(docker ps -q -l)
```

Finally we can now launch our cluster specifying the tcp-gossip stack with the location of the gossip router:

```
docker run -it jboss/infinispan-server -Djboss.default.
jgroups.stack=tcp-gossip -Djgroups.gossip.initial_
hosts=172.17.0.2[12001]
```

INTEGRATIONS

Infinispan can share data in lots of interesting and powerful ways.

HIBERNATE ORM

Infinispan can be a provider for Hibernate's second-level cache. Just add the *hibernate-infinispan* module as a dependency and specify the appropriate region factory in the *persistence.xml* file:

```
<property name="hibernate.cache.region.factory_class"
  value="org.hibernate.cache.infinispan.
  InfinispanRegionFactory" />
```

If you are using the WildFly application server, you only need to add the following to the *persistence.xml* file:

```
<property name="hibernate.cache.use_second_level_cache"
  value="true" />
```

APACHE HADOOP

The Hadoop Connector allows you to run your Hadoop jobs on data stored in Infinispan using InputFormat and OutputFormat. Here's how you'd set up a map/reduce job with Hadoop and Infinispan:

```
Configuration configuration = new Configuration();
String hosts = "172.17.0.2:11222;172.17.0.3:11222";
// Configures input/output caches
configuration.set(InfinispanConfiguration
  .INPUT_REMOTE_CACHE_SERVER_LIST, hosts);
configuration.set(InfinispanConfiguration
  .OUTPUT_REMOTE_CACHE_SERVER_LIST, hosts);
configuration.set(InfinispanConfiguration
  .INPUT_REMOTE_CACHE_NAME, "map-reduce-in");
configuration.set(InfinispanConfiguration
  .OUTPUT_REMOTE_CACHE_NAME, "map-reduce-out");

Job job = Job.getInstance(configuration, "Infinispan job");
// Map and Reduce implementation
job.setMapperClass(MapClass.class);
job.setReducerClass(ReduceClass.class);
job.setInputFormatClass(InfinispanInputFormat.class);
job.setOutputFormatClass(InfinispanOutputFormat.class);
```

APACHE SPARK

If you'd like to do the same but with Apache Spark, Infinispan comes with RDD and DStream integrations. The following Scala example creates a filtered RDD by supplying an Infinispan query:

```
val config = new Properties
config.put("infinispan.rdd.cacheName","my-cache")
config.put("infinispan.client.hotrod.server_list",
  "10.9.0.8:11222")
val infinispanRDD = new InfinispanRDD[String, MyEntity](sc,
  configuration = config)
// Assuming MyEntity is already stored in myCache
val query = Search.getQueryFactory(myCache)
  .from(classOf[MyEntity])
  .having("field").equal("some value")
  .toBuilder[RemoteQuery].build
val filteredRDD = rdd.filterByQuery(query, classOf[User])
```

APACHE CAMEL

Infinispan can also participate in Apache Camel routes acting as both a producer and consumer in both embedded and remote scenarios. Add the appropriate dependency via Maven:

```
<dependency>
  <groupId>org.apache.camel</groupId>
  <artifactId>camel-infinispan</artifactId>
  <version>x.x.x</version>
  <!-- use the same version as your Camel core version -->
</dependency>
```

Retrieving a value from a cache using a specific key:

```
from("direct:start")
  .setHeader(InfinispanConstants.OPERATION,
constant(InfinispanConstants.GET))
  .setHeader(InfinispanConstants.KEY, constant("123"))
  .to("infinispan://
localhost?cacheContainer=#cacheContainer");
```

ADDITIONAL RESOURCES

This Refcard only skims the surface of what can be achieved with Infinispan. For additional information, tutorials, examples, etc., please refer to the Infinispan website at infinispan.org.

ABOUT THE AUTHOR

TRISTAN TARRANT is the lead for the Infinispan project. Starting with a background in computer science specializing in parallel architectures, he has been involved in the design and implementation of software architectures for customers at all scales. An active contributor to JBoss Community and other open source projects for many years, he is a passionate advocate of open source and open standards.

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513
888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795

BROUGHT TO YOU IN PARTNERSHIP WITH
redhat