Configuring {brandname} 10.0

Table of Contents

Chapter 1. Configuration

{brandname} offers both declarative and programmatic configuration.

1.1. Configuring caches declaratively

Declarative configuration comes in a form of XML document that adheres to a provided {brandname} configuration XML schema.

Every aspect of {brandname} that can be configured declaratively can also be configured programmatically. In fact, declarative configuration, behind the scenes, invokes the programmatic configuration API as the XML configuration file is being processed. One can even use a combination of these approaches. For example, you can read static XML configuration files and at runtime programmatically tune that same configuration. Or you can use a certain static configuration defined in XML as a starting point or template for defining additional configurations in runtime.

There are two main configuration abstractions in {brandname}: `global` and `cache`.

Global configuration

Global configuration defines global settings shared among all cache instances created by a single EmbeddedCacheManager. Shared resources like thread pools, serialization/marshalling settings, transport and network settings, JMX domains are all part of global configuration.

Cache configuration

Cache configuration is specific to the actual caching domain itself: it specifies eviction, locking, transaction, clustering, persistence etc. You can specify as many named cache configurations as you need. One of these caches can be indicated as the `default` cache, which is the cache returned by the `CacheManager.getCache()` API, whereas other named caches are retrieved via the `CacheManager.getCache(String name)` API.

Whenever they are specified, named caches inherit settings from the default cache while additional behavior can be specified or overridden. {brandname} also provides a very flexible inheritance mechanism, where you can define a hierarchy of configuration templates, allowing multiple caches to share the same settings, or overriding specific parameters as necessary.

Embedded and Server configuration use different schemas, but we strive to maintain them as compatible as possible so that you can easily migrate between the two.

One of the major goals of {brandname} is to aim for zero configuration. A simple XML configuration file containing nothing more than a single infinispan element is enough to get you started. The configuration file listed below provides sensible defaults and is perfectly valid.

infinispan.xml

```
<infinispan />
```

However, that would only give you the most basic, local mode, non-clustered cache manager with no caches. Non-basic configurations are very likely to use customized global and default cache elements.

Declarative configuration is the most common approach to configuring {brandname} cache instances. In order to read XML configuration files one would typically construct an instance of DefaultCacheManager by pointing to an XML file containing {brandname} configuration. Once the configuration file is read you can obtain reference to the default cache instance.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

or any other named instance specified in `my-config-file.xml`.

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

The name of the default cache is defined in the `<cache-container>` element of the XML configuration file, and additional caches can be configured using the `<local-cache>`,`<distributed-cache>` ,`<invalidation-cache>` or `<replicated-cache>` elements.

The following example shows the simplest possible configuration for each of the cache types supported by {brandname}:

```
<infinispan>
   <cache-container default-cache="local">
      <transport cluster="mycluster"/>
      <local-cache name="local"/>
      <invalidation-cache name="invalidation" mode="SYNC"/>
      <replicated-cache name="repl-sync" mode="SYNC"/>
      <distributed-cache name="dist-sync" mode="SYNC"/>
   </cache-container>
</infinispan>
```

1.1.1. Cache configuration templates

As mentioned above, {brandname} supports the notion of *configuration templates*. These are full or partial configuration declarations which can be shared among multiple caches or as the basis for more complex configurations.

The following example shows how a configuration named `local-template` is used to define a cache named `local`.

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="local-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="local-template" />
   </cache-container>
</infinispan>
```

Templates can inherit from previously defined templates, augmenting and/or overriding some or all of the configuration elements:

```
<infinispan>
   <cache-container default-cache="local">
      <!-- template configurations -->
      <local-cache-configuration name="base-template">
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <local-cache-configuration name="extended-template" configuration="base-
template">
         <expiration lifespan="20"/>
         <memory>
            <object size="2000"/>
         </memory>
      </local-cache-configuration>

      <!-- cache definitions -->
      <local-cache name="local" configuration="base-template" />
      <local-cache name="local-bounded" configuration="extended-template" />
   </cache-container>
</infinispan>
```

In the above example, `base-template` defines a local cache with a specific *expiration* configuration. The `extended-template` configuration inherits from `base-template`, overriding just a single parameter of the *expiration* element (all other attributes are inherited) and adds a *memory* element. Finally, two caches are defined: `local` which uses the `base-template` configuration and `local-bounded` which uses the `extended-template` configuration.

Be aware that for multi-valued elements (such as `properties`) the inheritance is additive, i.e. the child configuration will be the result of merging the properties from the parent and its own.

1.1.2. Cache configuration wildcards

An alternative way to apply templates to caches is to use wildcards in the template name, e.g. `basecache*`. Any cache whose name matches the template wildcard will inherit that configuration.

```
<infinispan>
    <cache-container>
        <local-cache-configuration name="basecache*">
            <expiration interval="10500" lifespan="11" max-idle="11"/>
        </local-cache-configuration>
        <local-cache name="basecache-1"/>
        <local-cache name="basecache-2"/>
    </cache-container>
</infinispan>
```

Above, caches `basecache-1` and `basecache-2` will use the `basecache*` configuration. The configuration will also be applied when retrieving undefined caches programmatically.

If a cache name matches multiple wildcards, i.e. it is ambiguous, an exception will be thrown.

XInclude support

The configuration parser supports XInclude which means you can split your XML configuration across multiple files:

infinispan.xml

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
    <cache-container>
        <local-cache name="cache-1"/>
        <xi:include href="included.xml" />
    </cache-container>
</infinispan>
```

included.xml

```
<local-cache name="cache-1"/>
```

the parser supports a minimal subset of the XInclude spec (no support for XPointer, fallback, text processing and content negotiation).

1.1.3. Declarative configuration reference

For more details on the declarative configuration schema, refer to the configuration reference. If you are using XML editing tools for configuration writing you can use the provided {brandname} schema to assist you.

1.2. Configuring caches programmatically

Programmatic {brandname} configuration is centered around the CacheManager and ConfigurationBuilder API. Although every single aspect of {brandname} configuration could be set programmatically, the most usual approach is to create a starting point in a form of XML configuration file and then in runtime, if needed, programmatically tune a specific configuration to suit the use case best.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

Let's assume that a new synchronously replicated cache is to be configured programmatically. First, a fresh instance of Configuration object is created using ConfigurationBuilder helper object, and the cache mode is set to synchronous replication. Finally, the configuration is defined/registered with a manager.

```
Configuration c = new ConfigurationBuilder().clustering().cacheMode(CacheMode
.REPL_SYNC).build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

The default cache configuration (or any other cache configuration) can be used as a starting point for creation of a new cache. For example, lets say that `infinispan-config-file.xml` specifies a replicated cache as a default and that a distributed cache is desired with a specific L1 lifespan while at the same time retaining all other aspects of a default cache. Therefore, the starting point would be to read an instance of a default Configuration object and use `ConfigurationBuilder` to construct and modify cache mode and L1 lifespan on a new `Configuration` object. As a final step the configuration is defined/registered with a manager.

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = manager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering().cacheMode
(CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

As long as the base configuration is the default named cache, the previous code works perfectly fine. However, other times the base configuration might be another named cache. So, how can new configurations be defined based on other defined caches? Take the previous example and imagine that instead of taking the default cache as base, a named cache called "replicatedCache" is used as base. The code would look something like this:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache");
Configuration c = new ConfigurationBuilder().read(rc).clustering().cacheMode(
CacheMode.DIST_SYNC).l1().lifespan(60000L).build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Refer to CacheManager , ConfigurationBuilder , Configuration , and GlobalConfiguration javadocs for more details.

1.2.1. ConfigurationBuilder Programmatic Configuration API

While the above paragraph shows how to combine declarative and programmatic configuration, starting from an XML configuration is completely optional. The ConfigurationBuilder fluent interface style allows for easier to write and more readable programmatic configuration. This approach can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this API:

1.2.2. Configuring the transport

One of the most commonly configured global option is the transport layer, where you indicate how an {brandname} node will discover the others:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
        .addProperty("configurationFile", "jgroups-tcp.xml")
        .machineId("qa-machine").rackId("qa-rack")
      .build();
```

1.2.3. Using a custom JChannel

If you want to construct the JGroups JChannel by yourself, you can do so.

The JChannel must not be already connected.

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
JChannel jchannel = new JChannel();
// Configure the jchannel to your needs.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

1.2.4. Enabling JMX MBeans and statistics

Sometimes you might also want to enable collection of global JMX statistics at cache manager level or get information about the transport. To enable global JMX statistics simply do:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
  .enable()
  .build();
```

Please note that by not enabling (or by explicitly disabling) global JMX statistics your are just turning off statistics collection. The corresponding MBean is still registered and can be used to manage the cache manager in general, but the statistics attributes do not return meaningful values.

Further options at the global JMX statistics level allows you to configure the cache manager name which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookup(new JBossMBeanServerLookup())
  .build();
```

1.2.5. Configuring the thread pools

Some of the {brandname} features are powered by a group of the thread pool executors which can also be tweaked at this global level. For example:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
   .replicationQueueThreadPool()
     .threadPoolFactory(ScheduledThreadPoolExecutorFactory.create())
  .build();
```

You can not only configure global, cache manager level, options, but you can also configure cache level options such as the cluster mode:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Or you can configure eviction and expiration settings:

```
Configuration config = new ConfigurationBuilder()
           .memory()
             .size(20000)
          .expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

1.2.6. Configuring transactions and locking

An application might also want to interact with an {brandname} cache within the boundaries of JTA and to do that you need to configure the transaction layer and optionally tweak the locking settings. When interacting with transactional caches, you might want to enable recovery to deal with transactions that finished with an heuristic outcome and if you do that, you will often want to enable JMX management and statistics gathering too:

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
    .versioning().enable().scheme(VersioningScheme.SIMPLE)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
  .jmxStatistics()
  .build();
```

1.2.7. Configuring cache stores

Configuring {brandname} with chained cache stores is simple too:

```
Configuration config = new ConfigurationBuilder()
   .persistence().passivation(false)
   .addSingleFileStore().location("/tmp").async().enable()
   .preload(false).shared(false).threadPoolSize(20).build();
```

1.2.8. Advanced programmatic configuration

The fluent configuration can also be used to configure more advanced or exotic options, such as advanced externalizers:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(998, new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

Or, add custom interceptors:

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().addInterceptor()
    .interceptor(new FirstInterceptor()).position(InterceptorConfiguration.Position
.FIRST)
    .interceptor(new LastInterceptor()).position(InterceptorConfiguration.Position
.LAST)
    .interceptor(new FixPositionInterceptor()).index(8)
    .interceptor(new AfterInterceptor()).after(NonTransactionalLockingInterceptor
.class)
    .interceptor(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

For information on the individual configuration options, please check the configuration guide.

1.3. Configuration Migration Tools

The configuration format of {brandname} has changed since schema version 6.0 in order to align the embedded schema with the one used by the server. For this reason, when upgrading to schema 7.x or later, you should use the configuration converter included in the *all* distribution. Simply invoke it from the command-line passing the old configuration file as the first parameter and the name of the converted file as the second parameter.

Unix/Linux/macOS:

```
bin/config-converter.sh oldconfig.xml newconfig.xml
```

Windows:

```
bin\config-converter.bat oldconfig.xml newconfig.xml
```

If you wish to help write conversion tools from other caching systems, please contact infinispan-dev.

1.4. Clustered Configuration

{brandname} uses JGroups for network communications when in clustered mode. {brandname} ships with *pre-configured* JGroups stacks that make it easy for you to jump-start a clustered configuration.

1.4.1. Using an external JGroups file

If you are configuring your cache programmatically, all you need to do is:

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
   .transport().defaultTransport()
   .addProperty("configurationFile", "jgroups.xml")
   .build();
```

and if you happen to use an XML file to configure {brandname}, just use:

```
<infinispan>
  <jgroups>
     <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <transport stack="external-file" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>

  ...

</infinispan>
```

In both cases above, {brandname} looks for *jgroups.xml* first in your classpath, and then for an absolute path name if not found in the classpath.

1.4.2. Use one of the pre-configured JGroups files

{brandname} ships with a few different JGroups files (packaged in infinispan-core.jar) which means they will already be on your classpath by default. All you need to do is specify the file name, e.g., instead of `jgroups.xml` above, specify `/default-configs/default-jgroups-tcp.xml`.

The available configurations (with their stack names in brackets) are:

- default-jgroups-udp.xml (`udp`) - Uses UDP as a transport, and UDP multicast for discovery. Usually suitable for larger (over 100 nodes) clusters *or* if you are using replication or invalidation. Minimises opening too many sockets.
- default-jgroups-tcp.xml (`tcp`) - Uses TCP as a transport and UDP multicast for discovery. Better for smaller clusters (under 100 nodes) *only if* you are using distribution, as TCP is more efficient as a point-to-point protocol

- default-jgroups-ec2.xml (`ec2`) - Uses TCP as a transport and S3_PING for discovery. Suitable on Amazon EC2 nodes where UDP multicast is not available.
- default-jgroups-kubernetes.xml (`kubernetes`) - Uses TCP as a transport and DNS_PING for discovery. Suitable on Kubernetes and OpenShift nodes where UDP multicast is not always available.
- default-jgroups-google.xml (`google`) - Uses TCP as a transport and GOOGLE_PING2 for discovery. Suitable on Google Cloud Platform nodes where UDP multicast is not available.
- default-jgroups-azure.xml (`azure`) - Uses TCP as a transport and GOOGLE_PING2 for discovery. Suitable on Google Cloud Platform nodes where UDP multicast is not available.

When using the XML configuration, stack names for the above are pre-declared for you, so that you don't need to do it yourself. Just reference them directly in the transport:

```
<infinispan>
  <cache-container default-cache="replicatedCache">
    <transport stack="tcp" />

    ...

  </cache-container>
</infinispan>
```

1.5. Inline JGroups configurations

The Infinispan XML configuration also allows you to embed JGroups configurations without requiring an external file:

```
<infinispan>
  <jgroups>
    <stack name="prod">
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000" send_buf_size=
"640000"/>
      <MPING bind_addr="127.0.0.1" break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:228.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK />
      <FD_ALL timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="100" xmit_table_num_rows=
"50"
                      xmit_table_msgs_per_row="1024" xmit_table_max_compaction_time=
"30000" />
      <UNICAST3 xmit_interval="100" xmit_table_num_rows="50" xmit_table_msgs_per_row=
"1024"
                xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE stability_delay="200" desired_avg_gossip="2000" max_bytes="1M" />
      <pbcast.GMS print_local_addr="false" join_timeout="${jgroups.join_timeout:2000}"
/>
      <UFC_NB max_credits="3m" min_threshold="0.40" />
      <MFC_NB max_credits="3m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <transport stack="mystack" />

    ...

  </cache-container>
</infinispan>
```

JGroups stacks can be quite complex: most of the time, you only need to use a different discovery mechanism, add security, or make a slight tweak to a couple of parameters. For these cases, you can use stack inheritance to base your configuration off a known "good" JGroups stack. The following example creates a new `gossip-prod` based on the `prod` stack defined above, replacing the `MPING` protocol with `TCPGOSSIP`, increasing the `VERIFY_SUSPECT` timeout, removing the `FD_SOCK` protocol and adding `SYM_ENCRYPT`:

```
...

<jgroups>
  <stack name="gossip-prod" extends="prod">
    <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
             stack.combine="REPLACE" stack.position="MPING" />
    <FD_SOCK stack.combine="REMOVE"/>
    <VERIFY_SUSPECT timeout="2000"/>
    <SYM_ENCRYPT sym_algorithm="AES"
                 key_store_name="defaultStore.keystore"
                 store_password="changeit"
                 alias="myKey" stack.combine="INSERT_AFTER" stack.position=
"pbcast.NAKACK2" />
  </stack>
</jgroups>

...
```

The `stack.combine` attribute determines the type of the operation:

- `COMBINE`: overrides an existing protocol's attributes
- `REPLACE`: replaces an existing protocol, identified by the `stack.position` attribute. If this attribute is missing, it defaults to the same protocol, in which case all non-specified attributes are reset to their defaults.
- `INSERT_AFTER`: inserts a protocol after another protocol identified by the `stack.position` attribute.
- `REMOVE`: removes the protocol.

1.5.1. Tuning JGroups settings

The settings above can be further tuned without editing the XML files themselves. Passing in certain system properties to your JVM at startup can affect the behaviour of some of these settings. The table below shows you which settings can be configured in this way. E.g.,

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=10.11.12.13
```

Table 1. default-jgroups-udp.xml

System Property	*Description*	*Default*	*Required?*
jgroups.udp.mcast_addr	IP address to use for multicast (both for communications and discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No

jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Table 2. default-jgroups-tcp.xml

System Property	*Description*	*Default*	*Required?*
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.udp.mcast_addr	IP address to use for multicast (for discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Table 3. default-jgroups-ec2.xml

System Property	*Description*	*Default*	*Required?*
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.s3.access_key	The Amazon S3 access key used to access an S3 bucket		No
jgroups.s3.secret_access_key	The Amazon S3 secret key used to access an S3 bucket		No

jgroups.s3.bucket	Name of the Amazon S3 bucket to use. Must be unique and must already exist		No

Table 4. default-jgroups-kubernetes.xml

System Property	*Description*	*Default*	*Required?*
jgroups.tcp.address	IP address to use for the TCP transport.	eth0	No
jgroups.tcp.port	Port to use for TCP socket	7800	No

1.5.2. Further reading

JGroups also supports more system property overrides, details of which can be found on this page: SystemProps

In addition, the JGroups configuration files shipped with {brandname} are intended as a jumping off point to getting something up and running, and working. More often than not though, you will want to fine-tune your JGroups stack further to extract every ounce of performance from your network equipment. For this, your next stop should be the JGroups manual which has a detailed section on configuring each of the protocols you see in a JGroups configuration file.