Chapter 4. Document Structure

The following chapters explain the core functionality offered by Spring Data GemFire.

Bootstrapping GemFire through the Spring Container describes the configuration support provided for bootstrapping, configuring, initializing and accessing GemFire Caches, Cache Servers, Regions, and related Distributed System components.

Working with the GemFire APIs explains the integration between the GemFire APIs and the various data access features available in Spring, such as transaction management and exception translation.

Working with GemFire Serialization describes the enhancements for GemFire (de)serialization and management of associated objects.

POJO mapping describes persistence mapping for POJOs stored in GemFire using Spring Data.

GemFire Repositories describes how to create and use GemFire Repositories using Spring Data.

Annotation Support for Function Execution describes how to create and use GemFire Functions using annotations.

Bootstrapping a Spring ApplicationContext in GemFire describes how to bootstrap a Spring ApplicationContext running in a GemFire Server using Gfsh.

Sample Applications describes the samples provided with the distribution to illustrate the various features available in Spring Data GemFire.

Chapter 5. Bootstrapping GemFire through the Spring Container

Spring Data GemFire provides full configuration and initialization of the GemFire data grid through Spring's IoC container and provides several classes that simplify the configuration of GemFire components including Caches, Regions, WAN Gateways, Persistence Backup, and other Distributed System components to support a variety of scenarios with minimal effort.

NOTE This section assumes basic familiarity with GemFire. For more information see the product documentation.

5.1. Advantages of using Spring over GemFire `cache.xml`

As of release 1.2.0, Spring Data GemFire's XML namespace supports full configuration of the GemFire in-memory data grid. In fact, Spring Data GemFire's XML namespace is considered to be the preferred way to configure GemFire. GemFire will continue to support native `cache.xml` for legacy reasons, but GemFire application developers can now do everything in Spring XML and take advantage of the many wonderful things Spring has to offer such as modular XML configuration, property placeholders and overrides, SpEL, and environment profiles. Behind the XML namespace, Spring Data GemFire makes extensive use of Spring's `FactoryBean` pattern to simplify the creation, configuration and initialization of GemFire components.

For example, GemFire provides several callback interfaces, such as `CacheListener`, `CacheWriter`, and `CacheLoader`, that allow developers to add custom event handlers. Using Spring's IoC container, these callbacks may be configured as normal Spring beans and injected into GemFire components. This is a significant improvement over native `cache.xml`, which provides relatively limited configuration options and requires callbacks to implement GemFire's `Declarable` interface (see Wiring `Declarable` components to see how you can still use `Declarables` within Spring's IoC/DI container).

In addition, IDEs such as the Spring Tool Suite (STS) provide excellent support for Spring XML namespaces, such as code completion, pop-up annotations, and real time validation, making them easy to use.

5.2. Using the Core Spring Data GemFire Namespace

To simplify configuration, Spring Data GemFire provides a dedicated XML namespace for configuring core GemFire components. It is also possible to configure beans directly using Spring's standard <bean> definition. However, as of Spring Data GemFire 1.2.0, all bean properties are exposed via the XML namespace so there is little benefit to using raw bean definitions. For more information about XML Schema-based configuration in Spring, see this appendix in the Spring Framework reference documentation.

NOTE	Spring Data Repository support uses a separate XML namespace. See GemFire Repositories for more information on how to configure Spring Data GemFire Repositories.

To use the Spring Data GemFire XML namespace, simply declare it in your Spring XML configuration meta-data:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:gfe="http://www.springframework.org/schema/gemfire"① ②
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd"> ③

  <bean id ... >

  <gfe:cache ...> ④

</beans>
```

① Spring GemFire namespace prefix. Any name will do but through out the reference documentation, `gfe` will be used.

② The namespace URI.

③ The namespace URI location. Note that even though the location points to an external address (which exists and is valid), Spring will resolve the schema locally as it is included in the Spring Data GemFire library.

④ Declaration example for the GemFire namespace. Notice the prefix usage.

NOTE

It is possible to change the default namespace, for example from `beans` to `gfe`. This is useful for configuration composed mainly of GemFire components as it avoids declaring the prefix. To achieve this, simply swap the namespace prefix declaration above:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/gemfire" ①
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"②
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd">

  <beans:bean id ... > ③

  <cache ...> ④

</beans>
```

① The default namespace declaration for this XML file points to the Spring Data GemFire namespace.

② The `beans` namespace prefix declaration.

③ Bean declaration using the `beans` namespace. Notice the prefix.

④ Bean declaration using the `gfe` namespace. Notice the lack of prefix (as the default namespace is used).

5.3. Configuring a GemFire Cache

To use GemFire, a developer needs to either create a new `Cache` or connect to an existing one. With the current version of GemFire, there can be only one open Cache per VM (technically, per `ClassLoader`). In most cases, the `Cache` should only be created once.

NOTE

This section describes the creation and configuration of a cache member, appropriate in peer-to-peer topologies and cache servers. A cache member is also commonly used for standalone applications, integration tests and proof of concepts. In typical production systems, most application processes will act as cache clients, creating a `ClientCache` instance instead. This is described in the sections Configuring a GemFire ClientCache and Client Region.

A cache with default configuration can be created with a very simple declaration:

```
<gfe:cache/>
```

During Spring container initialization, any application context containing this cache definition will register a `CacheFactoryBean` that creates a Spring bean named `gemfireCache` referencing a GemFire `Cache` instance. This bean will refer to either an existing cache, or if one does not already exist, a newly created one. Since no additional properties were specified, a newly created cache will apply the default cache configuration.

All *Spring Data GemFire* components that depend on the cache respect this naming convention, so there is no need to explicitly declare the cache dependency. If you prefer, you can make the dependency explicit via the `cache-ref` attribute provided by various SDG namespace elements. Also, you can easily override the cache's bean name using the `id` attribute:

```
<gfe:cache id="my-cache"/>
```

Starting with *Spring Data GemFire* v1.2.0, a GemFire `Cache` can be fully configured using Spring. However, GemFire's native XML configuration file, `cache.xml`, is also supported. For situations in which the GemFire cache needs to be configured natively, simply provide a reference to the GemFire XML configuration file using the `cache-xml-location` attribute:

```
<gfe:cache id="cache-using-native-xml" cache-xml-location="classpath:cache.xml"/>
```

In this example, if the cache needs to be created, it will use the file named `cache.xml` located in the classpath root to configure it.

NOTE	The configuration makes use of Spring's `Resource` abstraction to locate the file. This allows various search patterns to be used, depending on the runtime environment or the prefix specified (if any) in the resource location.

In addition to referencing an external XML configuration file, a developer may also specify GemFire System properties using any of Spring's `Properties` support features.

For example, the developer may use the `properties` element defined in the `util` namespace to define `Properties` directly or load properties from a properties file:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:gfe="http://www.springframework.org/schema/gemfire"
  xmlns:util="http://www.springframework.org/schema/util"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
    http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

  <util:properties id="gemfireProperties" location=
"file:/pivotal/gemfire/gemfire.properties"/>

  <gfe:cache properties-ref="gemfireProperties"/>

</beans>
```

The latter approach is recommended for externalizing environment specific settings outside the application configuration.

NOTE	Cache settings apply only if a new cache needs to be created. If an open cache already exists in the VM, these settings are ignored.

5.3.1. Advanced Cache Configuration

For advanced cache configuration, the `cache` element provides a number of configuration options exposed as attributes or child elements:

```
①
<gfe:cache
    close="false"
    copy-on-read="true"
    critical-heap-percentage="70"
    eviction-heap-percentage="60"
    enable-auto-reconnect="false" ②
    lock-lease="120"
    lock-timeout="60"
    message-sync-interval="1"
    pdx-serializer-ref="myPdxSerializer"
    pdx-persistent="true"
    pdx-disk-store="diskStore"
    pdx-read-serialized="false"
    pdx-ignore-unread-fields="true"
    search-timeout="300"
    use-cluster-configuration="false" ③
    lazy-init="true">

  <gfe:transaction-listener ref="myTransactionListener"/> ④

  <gfe:transaction-writer> ⑤
    <bean class="org.springframework.data.gemfire.example.TransactionListener"/>
  </gfe:transaction-writer>

  <gfe:gateway-conflict-resolver ref="myGatewayConflictResolver"/> ⑥

  <gfe:dynamic-region-factory/> ⑦

  <gfe:jndi-binding jndi-name="myDataSource" type="ManagedDataSource"/> ⑧

</gfe:cache>
```

① Various cache options are supported by attributes. For further information regarding anything shown in this example, please consult the GemFire product documentation. The `close` attribute determines if the cache should be closed when the Spring application context is closed. The default is `true` however for cases in which multiple application contexts use the cache (common in web applications), set this value to `false`. The `lazy-init` attribute determines if the cache should be initialized before another bean references it. The default is `true` however in some cases it may be convenient to set this value to `false`.

② Setting the `enable-auto-reconnect` attribute to true (default is false), allows a disconnected GemFire member to automatically reconnect and rejoin a GemFire cluster. See the GemFire product documentation for more details.

③ Setting the `use-cluster-configuration` attribute to true (default is false) to enable a GemFire member to retrieve the common, shared Cluster-based configuration from a Locator. See the GemFire product documentation for more details.

④ An example of a `TransactionListener` callback declaration using a bean reference. The referenced bean must implement TransactionListener. `TransactionListener(s)` can be

implemented to handle transaction related events.

⑤ An example of a `TransactionWriter` callback declaration using an inner bean declaration this time. The bean must implement TransactionWriter. `TransactionWriter` is a callback that is allowed to veto a transaction.

⑥ An example of a `GatewayConflictResolver` declaration using a bean reference. The referenced bean must implement GatewayConflictResolver. GatewayConflictResolver is a Cache-level plugin that is called upon to decide what to do with events that originate in other systems and arrive through the WAN Gateway.

⑦ Enable GemFire's DynamicRegionFactory, which provides a distributed region creation service.

⑧ Declares a JNDI binding to enlist an external DataSource in a GemFire transaction.

NOTE

The `use-bean-factory-locator` attribute (not shown) deserves a mention. The factory bean responsible for creating the cache uses an internal Spring type called a `BeanFactoryLocator` to enable user classes declared in GemFire's native `cache.xml` to be registered as Spring beans. The `BeanFactoryLocator` implementation also permits only one bean definition for a cache with a given id. In certain situations, such as running JUnit integration tests from within Eclipse, it is necessary to disable the `BeanFactoryLocator` by setting this value to false to prevent an exception. This exception may also arise during JUnit tests running from a build script. In this case the test runner should be configured to fork a new JVM for each test (in maven, set `<forkmode>always</forkmode>`) . Generally, there is no harm in setting this value to false.

Enabling PDX Serialization

The example above includes a number of attributes related to GemFire's enhanced serialization framework, PDX. While a complete discussion of PDX is beyond the scope of this reference guide, it is important to note that PDX is enabled by registering a PDX serializer which is done via the `pdx-serializer` attribute. GemFire provides an implementation class `com.gemstone.gemfire.pdx.ReflectionBasedAutoSerializer`, however it is common for developers to provide their own implementation. The value of the attribute is simply a reference to a Spring bean that implements the required interface. More information on serialization support can be found in Working with GemFire Serialization

Enabling auto-reconnect

Setting the `<gfe:cache enable-auto-reconnect="[true|false*]>` attribute to true should be done with care.

Generally, enabling 'auto-reconnect' should only be done in cases where *Spring Data GemFire's* XML namespace is used to configure and bootstrap a new GemFire Server data node to add to the cluster. In other words, 'auto-reconnect' should not be used when *Spring Data GemFire* is used to develop and build an GemFire application that also happens to be a peer cache member of the GemFire cluster.

The main reason is most GemFire applications use references to the GemFire cache or regions in order to perform data access operations. The references are "injected" by the Spring container into

application components (e.g. DAOs or Repositories) for use by the application. When a member (such as the application) is forcefully disconnected from the rest of the cluster, presumably because the member (the application) has become unresponsive for a period of time, or network partition separates one or more members (along with the application peer cache member) into a group that is too small to act as the distributed system, the member will shutdown and all GemFire component references (e.g. Cache, Regions, etc) become invalid.

Essentially, the current forced-disconnect processing in each member dismantles the system from the ground up. It shuts down the JGroups stack, puts the Distributed System in a shut-down state and then closes the Cache. This effectively loses all in-memory information.

After being disconnected from a distributed system and successfully shutting down, the GemFire member then restarts in a "reconnecting" state, while periodically attempting to rejoin the distributed system. If the member succeeds in reconnecting, the member rebuilds its "view" of the distributed system from existing members and receives a new distributed system ID.

This means the cache, regions and other GemFire components are reconstructed and all old references that may have been injected into application are now stale and no longer valid.

GemFire makes no guarantee, even when using the GemFire public Java API, that application cache, region or other component references will be automatically refreshed by the reconnect operation. As such, applications must take care to refresh their own references.

Unfortunately there is no way to be "notified" of a disconnect and subsequently a reconnect event. If so, the application developer would then have a clean way to know when to call ConfigurableApplicationContext.refresh(), if even applicable for an application to do so, which is why this "feature" of GemFire 8 is not recommended for peer cache GemFire applications.

For more information about 'auto-reconnect', see GemFire's product documentation.

Using Cluster-based Configuration

GemFire 8's new Cluster-based Configuration Service is a convenient way for a member joining the cluster to get a "consistent view" of the cluster, by using the shared, persistent configuration maintained by a Locator, ensuring the member's configuration will be compatible with the GemFire distributed system when the member joins.

This feature of Spring Data GemFire (setting the `use-cluster-configuration` attribute to true) works in the same way as the `cache-xml-location` attribute, except the source of the GemFire configuration meta-data comes from a network Locator as opposed to a native `cache.xml` file.

All GemFire native configuration meta-data, whether from `cache.xml` or from the Cluster Configuration Service, gets applied before any Spring XML configuration meta-data. As such, Spring's config serves to "augment" the native GemFire configuration meta-data, which would most likely be specific to the application.

Again, to enable this feature, just specify the following in the Spring XML config:

```
<gfe:cache use-cluster-configuration="true"/>
```

NOTE	While certain GemFire tools, like Gfsh, have their actions "recorded" when any schema-like change is made (e.g. `gfsh>create region --name=Example --type=PARTITION`) to the cluster, Spring Data GemFire's configuration meta-data specified with the XML namespace is not recorded. The same is true when using GemFire's public Java API directly; it too is not recorded.

For more information on GemFire's Cluster Configuration Service, see the product documentation.

5.3.2. Configuring a GemFire Cache Server

Spring Data GemFire includes dedicated support for configuring a CacheServer, allowing complete configuration through the Spring container:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:gfe="http://www.springframework.org/schema/gemfire"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <gfe:cache />

  <!-- Advanced example depicting various cache server configuration options -->
  <gfe:cache-server id="advanced-config" auto-startup="true"
       bind-address="localhost" host-name-for-clients="localhost" port=
"${gemfire.cache.server.port}"
       load-poll-interval="2000" max-connections="22" max-message-count="1000"
       max-threads="16" max-time-between-pings="30000"
       groups="test-server">
    <gfe:subscription-config eviction-type="ENTRY" capacity="1000" disk-store=
"file://${java.io.tmpdir}"/>
  </gfe:cache-server>

   <context:property-placeholder location="classpath:cache-server.properties"/>

</beans>
```

The configuration above illustrates the `cache-server` element and the many options available.

NOTE

Rather than hard-coding the port, this configuration uses Spring's context namespace to declare a `property-placeholder`. property placeholder reads one or more properties files and then replaces property placeholders with values at runtime. This allows administrators to change values without having to touch the main application configuration. Spring also provides the SpEL and the environment abstraction to support externalization of environment-specific properties from the main codebase, easing deployment across multiple machines.

NOTE

To avoid initialization problems, the `CacheServer` started by *Spring Data GemFire* will start **after** the container has been fully initialized. This allows potential regions, listeners, writers or instantiators defined declaratively to be fully initialized and registered before the server starts accepting connections. Keep this in mind when programmatically configuring these elements as the server might start after your components and thus not be seen by the clients connecting right away.

5.3.3. Configuring a GemFire ClientCache

In addition to defining a GemFire peer Cache, *Spring Data GemFire* also supports the definition of a GemFire ClientCache in a Spring context. A `ClientCache` definition is very similar in configuration and use to the GemFire peer Cache and is supported by the `org.springframework.data.gemfire.client.ClientCacheFactoryBean`.

The simplest definition of a GemFire cache client with default configuration can be accomplished with the following declaration:

```
<beans>
    <gfe:client-cache />
</beans>
```

`client-cache` supports much of the same options as the cache element. However, as opposed to a **full-fledged** cache member, a client cache connects to a remote cache server through a Pool. By default, a Pool is created to connect to a server running on `localhost`, listening to port `40404`. The default Pool is used by all client Regions unless the Region is configured to use a different Pool.

Pools can be defined with the `pool` element. This client-side Pool can be used to configure connectivity directly to a server for individual entities or to the entire cache through one or more Locators.

For example, to customize the default Pool used by the `client-cache`, the developer needs to define a Pool and wire it to the cache definition:

```
<beans>
  <gfe:client-cache id="my-cache" pool-name="my-pool"/>

  <gfe:pool id="my-pool" subscription-enabled="true">
     <gfe:locator host="${gemfire.locator.host}" port="${gemfire.locator.port}"/>
 </gfe:pool>
</beans>
```

The `<client-cache>` element also includes the `ready-for-events` attribute. If set to `true`, the client cache initialization will include a call to ClientCache.readyForEvents().

Client-side configuration is covered in more detail in Client Region.

GemFire's DEFAULT Pool and Spring Data GemFire Pool Definitions

If a GemFire `ClientCache` is local-only, then no Pool definition is required. For instance, a developer may define:

```
<gfe:client-cache/>

<gfe:client-region id="Example" shortcut="LOCAL"/>
```

In this case, the "Example" Region is `LOCAL` and no data is distributed between the client and a server, therefore, no Pool is necessary. This is true for any client-side, local-only Region, as defined by the GemFire's ClientRegionShortcut (all `LOCAL_*` shortcuts).

However, if the client Region is a (caching) proxy to a server-side Region, then a Pool is required. There are several ways to define and use a Pool in this case.

When a client cache, Pool and proxy-based Region are all defined, but not explicitly identified, *Spring Data GemFire* will resolve the references automatically for you.

For example:

```
<gfe:client-cache/>

<gfe:pool>
  <gfe:locator host="${gemfire.locator.host}" port="${gemfire.locator.port}"/>
</gfe:pool>

<gfe:client-region id="Example" shortcut="PROXY"/>
```

In this case, the client cache is identified as `gemfireCache`, the Pool as `gemfirePool` and the client Region as, well, "Example". However, the client cache will initialize GemFire's DEFAULT Pool from the `gemfirePool` and the client Region will use the `gemfirePool` when distributing data between the client and the server.

Spring Data GemFire basically resolves the above configuration to the following:

```
<gfe:client-cache id="gemfireCache" pool-name="gemfirePool"/>

<gfe:pool id="gemfirePool">
  <gfe:locator host="${gemfire.locator.host}" port="${gemfire.locator.port}"/>
</gfe:pool>

<gfe:client-region id="Example" cache-ref="gemfireCache" pool-name="gemfirePool"
shortcut="PROXY"/>
```

GemFire still creates a Pool called "DEFAULT". *Spring Data GemFire* will just cause the "DEFAULT" Pool to be initialized from `gemfirePool`. This is useful in situations where multiple Pools are defined and client Regions are using separate Pools.

Consider the following:

```
<gfe:client-cache pool-name="locatorPool"/>

<gfe:pool id="locatorPool">
  <gfe:locator host="${gemfire.locator.host}" port="${gemfire.locator.port}"/>
</gfe:pool>

<gfe:pool id="serverPool">
  <gfe:locator host="${gemfire.server.host}" port="${gemfire.server.port}"/>
</gfe:pool>

<gfe:client-region id="Example" pool-name="serverPool" shortcut="PROXY"/>

<gfe:client-region id="AnotherExample" shortcut="CACHING_PROXY"/>

<gfe:client-region id="YetAnotherExample" shortcut="LOCAL"/>
```

In this setup, the GemFire client cache's "DEFAULT" Pool is initialized from "locatorPool" as specified with the `pool-name` attribute. There is no *Spring Data GemFire*-defined `gemfirePool` since both Pools were explicitly identified (named) "locatorPool" and "serverPool", respectively.

The "Example" Region explicitly refers to and uses the "serverPool" exclusively. The "AnotherExample" Region uses GemFire's "DEFAULT" Pool, which was configured from the "locatorPool" based on the client cache bean definition's `pool-name` attribute.

Finally, the "YetAnotherExample" Region will not use a Pool since it is `LOCAL`.

NOTE	The "AnotherExample" Region would first look for a Pool bean named `gemfirePool`, but that would require the definition of an anonymous Pool bean (i.e. `<gfe:pool/>`) or a Pool bean explicitly named `gemfirePool` (e.g. `<gfe:pool id="gemfirePool"/>`).

NOTE	We could have either named "locatorPool", "gemfirePool", or made the Pool bean definition anonymous and it would have the same effect as the above configuration.

5.4. Using the GemFire Data Access Namespace

In addition to the core `gfe` namespace, Spring Data GemFire provides a `gfe-data` namespace intended primarily to simplify the development of GemFire client applications. This namespace currently supports for GemFire repositories and function execution and a `<datasource>` tag that offers a convenient way to connect to the data grid.

5.4.1. An Easy Way to Connect to GemFire

For many applications, A basic connection to a GemFire grid, using default values is sufficient. Spring Data GemFire's `<datasource>` tag provides a simple way to access data. The data source creates a client cache and connection pool. In addition, it will query the member servers for all existing root regions and create a proxy (empty) client region for each one.

```
<gfe-data:datasource>
   <locator host="somehost" port="1234"/>
</gfe-data:datasource>
```

The datasource tag is syntactically similar to `<gfe:pool>`. It may be configured with one or more locator or server tags to connect to an existing data grid. Additionally, all attributes available to configure a pool are supported. This configuration will automatically create ClientRegion beans for each region defined on members connected to the locator, so they may be seamlessly referenced by Spring Data mapping annotations, GemfireTemplate, and wired into application classes.

Of course, you can explicitly configure client regions. For example, if you want to cache data in local memory:

```
<gfe-data:datasource>
   <locator host="somehost" port="1234"/>
</gfe-data:datasource>

<gfe:client-region id="Customer" shortcut="CACHING_PROXY"/>
```

5.5. Configuring a GemFire Region

A region is required to store and retrieve data from the cache. `Region` is an interface extending `java.util.Map` and enables basic data access using familiar key-value semantics. The `Region` interface is wired into classes that require it so the actual region type is decoupled from the programming model . Typically each region is associated with one domain object, similar to a table in a relational database.

GemFire implements the following types of regions:

- **Replicated** - Data is replicated across all cache members that define the region. This provides very high read performance but writes take longer to perform the replication.
- **Partioned** - Data is partitioned into buckets among cache members that define the region. This provides high read and write performance and is suitable for very large data sets that are too big for a single node.
- **Local** - Data only exists on the local node.
- **Client** - Technically a client region is a local region that acts as a proxy to a replicated or partitioned region hosted on cache servers. It may hold data created or fetched locally. Alternately, it can be empty. Local updates are synchronized to the cache server. Also, a client region may subscribe to events in order to stay synchronized with changes originating from remote processes that access the same region.

For more information about the various region types and their capabilities as well as configuration options, please refer to the GemFire Developer's Guide and community site.

5.5.1. Using an externally configured Region

For referencing Regions already configured through GemFire `cache.xml` file, use the `lookup-region` element. Simply declare the target Region name with the`name` attribute; for example, to declare a bean definition named `region-bean` for an existing region named `Orders` one can use the following bean definition:

```
<gfe:lookup-region id="region-bean" name="Orders"/>
```

If the `name` is not specified, the bean's `id` will be used. The example above becomes:

```
<!-- lookup for a region called 'Orders' -->
<gfe:lookup-region id="Orders"/>
```

NOTE If the Region does not exist, an initialization exception will be thrown. For configuring new GemFire Regions, proceed to the appropriate sections below.

Note, in the previous examples, since no cache name was defined, the default naming convention (`gemfireCache`) was used. Alternately, one can reference the cache bean through the `cache-ref` attribute:

```
<gfe:cache id="cache"/>
<gfe:lookup-region id="region-bean" name="Orders" cache-ref="cache"/>
```

`lookup-region` provides a simple way of retrieving existing, pre-configured Regions without exposing the Region semantics or setup infrastructure.

5.5.2. Auto Region Lookup

New, as of Spring Data GemFire 1.5, is the ability to "auto-lookup" all Regions defined in GemFire's native cache.xml file, and imported into Spring config using the `cache-xml-location` attribute on the `<gfe:cache>` element in the GFE XML namespace.

For instance, given a GemFire `cache.xml` file of...

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC  "-//GemStone Systems, Inc.//GemFire Declarative Caching
7.0//EN"
    "http://www.gemstone.com/dtd/cache7_0.dtd">
<cache>
    <region name="Parent" refid="REPLICATE">
        <region name="Child" refid="REPLICATE"/>
    </region>
</cache>
```

A user may import the `cache.xml` file as follows...

```
<gfe:cache cache-xml-location="cache.xml"/>
```

A user can then use the `<gfe:lookup-region>` element (e.g. `<gfe:lookup-region id="Parent"/>`) to reference specific GemFire Regions as beans in the Spring context, or the user may choose to import all GemFire Regions defined in `cache.xml` with the new...

```
<gfe:auto-region-lookup/>
```

Spring Data GemFire will automatically create Spring beans referencing all GemFire Regions defined in `cache.xml` that have not been explicitly added to the Spring context with `<gfe:lookup-region>` bean declarations.

It is important to realize that Spring Data GemFire uses a Spring BeanPostProcessor to post process the Cache after it is both created and initialized to determine the Regions defined in GemFire to add as beans in the Spring context.

You may inject these "auto-looked-up" Regions like any other bean defined in the Spring context with 1 exception; you may need to define a `depends-on` association with the 'gemfireCache' bean as follows...

```
package example;

import ...

@Repository("appDao")
@DependsOn("gemfireCache")
public class ApplicationDao extends DaoSupport {

  @Resource(name = "Parent")
  private Region<?, ?> parent;

  @Resource(name = "/Parent/Child")
  private Region<?, ?> child;

  ...
}
```

The above Java example is applicable when using the Spring context's `component-scan` functionality.

If you are declaring your components using Spring XML, then you would do...

```
<bean class="example.ApplicationDao" depends-on="gemfireCache"/>
```

This ensures the GemFire Cache and all the Regions defined in `cache.xml` get created before any components with auto-wire references when using the new `<gfe:auto-region-lookup>` element.

5.5.3. Configuring Regions

Spring Data GemFire provides comprehensive support for configuring any type of GemFire Region via the following elements:

- Local Region `<local-region>`
- Replicated Region `<replicated-region>`
- Partitioned Region `<partitioned-region>`
- Client Region `<client-region>`

For a comprehensive description of Region types please consult the GemFire product documentation.

Common Region Attributes

The following table(s) list attributes available for various region types:

Table 1. Common Region Attributes

Name	Values	Description
cache-ref	**GemFire Cache bean name**	The name of the bean defining the GemFire Cache (by default 'gemfireCache').
close	**boolean, default:false (Note: The default was true prior to 1.3.0)**	Indicates whether the Region should be closed at shutdown.
cloning-enabled	**boolean, default:false**	When true, the updates are applied to a clone of the value and then the clone is saved to the cache. When false, the value is modified in place in the cache.
concurrency-checks-enabled	**boolean, default:true**	Determines whether members perform checks to provide consistent handling for concurrent or out-of-order updates to distributed Regions.
data-policy	**See GemFire's Data Policy**	The Region's Data Policy. Note, not all Data Policies are supported for every Region type.
destroy	**boolean, default:false**	Indicates whether the Region should be destroyed at shutdown.
disk-store-ref	**The name of a configured Disk Store.**	A reference to a bean created via the `disk-store` element.
disk-synchronous	**boolean, default:true**	Indicates whether Disk Store writes are synchronous.
enable-gateway	**boolean, default:false**	Indicates whether the Region will synchronize entries over a WAN Gateway.
hub-id	**The name of the Gateway Hub.**	This will automatically set enable-gateway to true. If enable-gateway is explicitly set to false, an exception will be thrown.
id	**Any valid bean name.**	Will also be the Region name by default.
ignore-if-exists	**boolean, default:false**	Ignores this bean definition configuration if the Region already exists in the GemFire Cache, resulting in a lookup instead.
ignore-jta	**boolean, default:false**	Indicates whether the Region participates in JTA transactions.
index-update-type	**synchronous or asynchronous, default:synchronous**	Indicates whether indices will be updated synchronously or asynchronously on entry creation.
initial-capacity	**integer, default:16**	The initial memory allocation for number of Region entries.
key-constraint	**Any valid, fully-qualified Java class name.**	The expected key type.

Name	Values	Description
load-factor	**float, default:.75**	Sets the initial parameters on the underlying java.util.ConcurrentHashMap used for storing Region entries.
name	**Any valid Region name.**	The name of the Region definition. If not specified, it will assume the value of the id attribute (the bean name).
persistent	**boolean, default:false**	Indicates whether the Region persists entries to a Disk Store (disk).
shorcut	*See http://data-docs-samples.cfapps.io/docs-gemfire/latest/javadocs/japi/com/gemstone/gemfire/cache/RegionShortcut.html	The RegionShortcut for this Region. Allows easy initialization of the region based on pre-defined defaults.
statistics	**boolean, default:false**	Indicates whether the Region reports statistics.
template	**The name of a Region Template.**	A reference to a bean created via one of the `*region-template` elements.
value-constraint	**Any valid, fully-qualified Java class name.**	The expected value type.

Cache Listeners

`CacheListeners` are registered with a Region to handle Region events such as entries being created, updated, destroyed, etc. A `CacheListener` can be any bean that implements the `CacheListener` interface. A Region may have multiple listeners, declared using the `cache-listener` element enclosed in a `*-region` element.

In the example below, there are two `CacheListener's` declared. The first references a top-level named Spring bean; the second is an anonymous inner bean definition.

```
<gfe:replicated-region id="region-with-listeners">
    <gfe:cache-listener>
        <!-- nested cache listener reference -->
        <ref bean="c-listener"/>
        <!-- nested cache listener declaration -->
        <bean class="some.pkg.AnotherSimpleCacheListener"/>
    </gfe:cache-listener>

    <bean id="c-listener" class="some.pkg.SimpleCacheListener"/>
</gfe:replicated-region>
```

The following example uses an alternate form of the `cache-listener` element with a `ref` attribute. This allows for more concise configuration for a single cache listener. Note that the namespace only allows a single `cache-listener` element so either the style above or below must be used.

WARNING	Using `ref` and a nested declaration in a `cache-listener`, or similar element, is illegal. The two options are mutually exclusive and using both on the same element will result in an exception.

```
<beans>
    <gfe:replicated-region id="region-with-one listener">
        <gfe:cache-listener ref="c-listener"/>
    </gfe:replicated-region>

    <bean id="c-listener" class="some.pkg.SimpleCacheListener"/>
 </beans>
```

NOTE	*Bean Reference Conventions* The `cache-listener` element is an example of a common pattern used in the namespace anywhere GemFire provides a callback interface to be implemented in order to invoke custom code in response to Cache or Region events. Using Spring's IoC container, the implementation is a standard Spring bean. In order to simplify the configuration, the schema allows a single occurrence of the `cache-listener` element, but it may contain nested bean references and inner bean definitions in any combination if multiple instances are permitted. The convention is to use the singular form (i.e., `cache-listener` vs `cache-listeners`) reflecting that the most common scenario will in fact be a single instance. We have already seen examples of this pattern in the advanced cache configuration example.

Cache Loaders and Cache Writers

Similar to `cache-listener`, the namespace provides `cache-loader` and `cache-writer` elements to register these respective components for a Region. A `CacheLoader` is invoked on a cache miss to allow an entry to be loaded from an external data source, a database for example. A `CacheWriter` is invoked before an entry is created or updated, intended for synchronizing to an external data source. The difference is GemFire only supports at most a single instance of each for each Region. However, either declaration style may be used.

See `CacheLoader` and `CacheWriter` for more details.

Subregions

In Release 1.2.0, Spring Data GemFire added support for subregions, allowing regions to be arranged in a hierarchical relationship. For example, GemFire allows for a **/Customer/Address** region and a different **/Employee/Address** region. Additionally, a subregion may have it's own subregions and its own configuration. A subregion does not inherit attributes from the parent region. Regions types may be mixed and matched subject to GemFire constraints. A subregion is naturally declared as a child element of a region. The subregion's name attribute is the simple name. The above example might be configured as: [source,nonxml]

```
<beans>

    <gfe:replicated-region name="Customer">
        <gfe:replicated-region name="Address"/>
    </gfe:replicated-region>

    <gfe:replicated-region name="Employee">
        <gfe:replicated-region name="Address"/>
    </gfe:replicated-region>

</beans>
```

Note that the `Monospaced ([id])` attribute is not permitted for a subregion. The subregions will be created with bean names **/Customer/Address** and **/Employee/Address**, respectively. So they may be injected using the full path name into other beans that use them, such as `GemfireTemplate`. The full path should also be used in OQL query strings.

5.5.4. Region Templates

Also new as of Spring Data GemFire 1.5 is Region Templates. This feature allows developers to define common Region configuration settings and attributes once and reuse the configuration among many Region bean definitions declared in the Spring context.

Spring Data GemFire introduces 5 new tags to the SDG XML namespace (XSD):

Table 2. Region Template Tags

Tag Name	Description
`<gfe:region-template>`	Defines common, generic Region attributes; extends `regionType` in the SDG 1.5 XSD
`<gfe:local-region-template>`	Defines common, 'Local' Region attributes; extends `localRegionType` in the SDG 1.5 XSD
`<gfe:partitioned-region-template>`	Defines common, 'PARTITION' Region attributes; extends `partitionedRegionType` in the SDG 1.5 XSD
`<gfe:replicated-region-template>`	Defines common, 'REPLICATE' Region attributes; extends `replicatedRegionType` in the SDG 1.5 XSD
`<gfe:client-region-template>`	Defines common, 'Client' Region attributes; extends `clientRegionType` in the SDG 1.5 XSD

In addition to the new tags, `<gfe:*-region>` elements along with the `<gfe:*-region-template>` elements have a `template` attribute used to define the Region Template from which to inherit the Region configuration. Even Region templates may inherit from other Region Templates.

Here is an example of 1 possible configuration…

```
<gfe:async-event-queue id="AEQ" persistent="false" parallel="false" dispatcher-
threads="4">
  <gfe:async-event-listener>
```

```
    <bean class="example.AeqListener"/>
  </gfe:async-event-listener>
</gfe:async-event-queue>

<gfe:region-template id="BaseRegionTemplate" cloning-enabled="true"
    concurrency-checks-enabled="false" disk-synchronous="false"
    ignore-jta="true" initial-capacity="51" key-constraint="java.lang.Long"
    load-factor="0.85" persistent="false" statistics="true"
    value-constraint="java.lang.String">
  <gfe:cache-listener>
    <bean class="example.CacheListenerOne"/>
    <bean class="example.CacheListenerTwo"/>
  </gfe:cache-listener>
  <gfe:entry-ttl timeout="300" action="INVALIDATE"/>
  <gfe:entry-tti timeout="600" action="DESTROY"/>
</gfe:region-template>

<gfe:region-template id="ExtendedRegionTemplate" template="BaseRegionTemplate"
    index-update-type="asynchronous" cloning-enabled="false"
    concurrency-checks-enabled="true" key-constraint="java.lang.Integer"
    load-factor="0.55">
  <gfe:cache-loader>
    <bean class="example.CacheLoader"/>
  </gfe:cache-loader>
  <gfe:cache-writer>
    <bean class="example.CacheWriter"/>
  </gfe:cache-writer>
  <gfe:membership-attributes required-roles="readWriteNode" loss-action="limited-
access" resumption-action="none"/>
  <gfe:async-event-queue-ref bean="AEQ"/>
</gfe:region-template>

<gfe:partitioned-region-template id="PartitionRegionTemplate" template=
"ExtendedRegionTemplate"
    copies="1" local-max-memory="1024" total-max-memory="16384" recovery-delay="60000"
    startup-recovery-delay="15000" enable-async-conflation="false"
    enable-subscription-conflation="true" load-factor="0.70"
    value-constraint="java.lang.Object">
  <gfe:partition-resolver>
    <bean class="example.PartitionResolver"/>
  </gfe:partition-resolver>
  <gfe:eviction type="ENTRY_COUNT" threshold="8192000" action="OVERFLOW_TO_DISK"/>
</gfe:partitioned-region-template>

<gfe:partitioned-region id="TemplateBasedPartitionRegion" template=
"PartitionRegionTemplate"
    copies="2" local-max-memory="8192" total-buckets="91" disk-synchronous="true"
    enable-async-conflation="true" ignore-jta="false" key-constraint="java.util.Date"
    persistent="true">
  <gfe:cache-writer>
    <bean class="example.CacheWriter"/>
```

```
  </gfe:cache-writer>
  <gfe:membership-attributes required-roles="admin,root" loss-action="no-access"
resumption-action="reinitialize"/>
  <gfe:partition-listener>
    <bean class="example.PartitionListener"/>
  </gfe:partition-listener>
  <gfe:subscription type="ALL"/>
</gfe:partitioned-region>
```

Region Templates will even work for Subregions. Notice that 'TemplateBasedPartitionRegion' extends 'PartitionRegionTemplate' which extends 'ExtendedRegionTemplate' which extends 'BaseRegionTemplate'. Attributes and sub-elements defined in subsequent, inherited Region bean definitions override what is in the parent.

Under-the-hood…

Spring Data GemFire applies Region Templates when the Spring application context configuration meta-data is **parsed**, and therefore, must be declared in the order of inheritance, in other words, parent templates before children. This ensures the proper configuration is applied, especially when element attributes or sub-elements are "overridden".

IMPORTANT	It is equally important to remember the Region types must only inherit from other similar typed Regions. For instance, it is not possible for a `<gfe:replicated-region>` to inherit from a `<gfe:partitioned-region-template>`.

NOTE	Region Templates are single-inheritance.

5.5.5. A Word of Caution on Regions, Subregions and Lookups

Prior to Spring Data GemFire 1.4, one of the underlying properties of the high-level `replicated-region`, `partitioned-region`, `local-region` and `client-region` elements in Spring Data GemFire's XML namespace, which correspond to GemFire's Region types based on Data Policy, is that these elements perform a lookup first before attempting to create the region. This is done in case the region already exists, which might be the case if the region was defined in GemFire's native configuration, e.g. `cache.xml`, thereby avoiding any errors. This was by design, though subject to change.

WARNING	The Spring team highly recommends that the `replicated-region`, `partitioned-region`, `local-region` and `client-region` elements be strictly used only for defining new regions. One of the problems with these elements doing a lookup first is, if the developer assumed that defining a bean definition for a REPLICATE region would create a new region, however, consequently a region with the same name already exists having different semantics for eviction, expiration, subscription and/or other attributes, this could adversely affect application logic and/or expectations thereby violating application requirements.

IMPORTANT	Recommended Practice - Only use the `replicated-region`, `partitioned-region`, `local-region` and `client-region` XML namespace elements for defining new regions.

However, because the high-level region elements perform a lookup first, this can cause problems for dependency injected region resources to application code, like DAOs or Repositories.

Take for instance the following native GemFire configuration file (e.g. `cachel.xml`)...

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cache PUBLIC  "-//GemStone Systems, Inc.//GemFire Declarative Caching
7.0//EN"
"http://www.gemstone.com/dtd/cache7_0.dtd">
<cache>
    <region name="Customers" refid="REPLICATE">
        <region name="Accounts" refid="REPLICATE">
            <region name="Orders" refid="REPLICATE">
                <region name="Items" refid="REPLICATE"/>
            </region>
        </region>
    </region>
</cache>
```

Also, consider that you might have defined a DAO as follows...

```
public class CustomerAccountDao extends GemDaoSupport {

    @Resource(name = "Customers/Accounts")
    private Region customersAccounts;

    ...
}
```

Here, we are injecting a reference to the `Customers/Accounts` GemFire Region in our DAO. As such, it is not uncommon for a developer to define beans for all or some of these regions in Spring XML configuration meta-data as follows...

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
          xmlns:gfe="http://www.springframework.org/schema/gemfire"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="
            http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd">

    <gfe:cache cache-xml-location="classpath:cache.xml"/>

    <gfe:lookup-region name="Customers/Accounts"/>
    <gfe:lookup-region name="Customers/Accounts/Orders"/>
</beans>
```

Here the `Customers/Accounts` and `Customers/Accounts/Orders` GemFire Regions are referenced as beans in the Spring context as "Customers/Accounts" and "Customers/Accounts/Orders", respectively. The nice thing about using the `lookup-region` element and the corresponding syntax above is that it allows a developer to reference a subregion directly without unnecessarily defining a bean for the parent region (e.g. `Customers`).

However, if now the developer changes his/her configuration meta-data syntax to using the nested format, like so...

```
<gfe:lookup-region name="Customers">
    <gfe:lookup-region name="Accounts">
        <gfe:lookup-region name="Orders"/>
    </gfe:lookup-region>
</gfe:lookup-region>
```

Or, perhaps the developer erroneously chooses to use the high-level `replicated-region` element, which will do a lookup first, as in...

```
<gfe:replicated-region name="Customers" persistent="true">
    <gfe:replicated-region name="Accounts" persistent="true">
        <gfe:replicated-region name="Orders" persistent="true"/>
    </gfe:replicated-region>
</gfe:replicated-region>
```

Then the region beans defined in the Spring context will consist of the following: `{ "Customers", "/Customers/Accounts", "/Customers/Accounts/Orders" }`. This means the dependency injected reference (i.e. `@Resource(name = "Customers/Accounts"))` is now broken since no bean with name "Customers/Accounts" is defined.

GemFire is flexible in referencing both parent regions and subregions. The parent can be referenced as "/Customers" or "Customers" and the child as "/Customers/Accounts" or just

"Customers/Accounts". However, Spring Data GemFire is very specific when it comes to naming beans after regions, typically always using the forward slash (/) to represents subregions (e.g. "/Customers/Accounts").

Therefore, it is recommended that users use either the nested `lookup-region` syntax as illustrated above, or define direct references with a leading forward slash (/) like so...

```
<gfe:lookup-region name="/Customers/Accounts"/>
<gfe:lookup-region name="/Customers/Accounts/Orders"/>
```

The example above where the nested `replicated-region` elements were used to reference the subregions serves to illustrate the problem stated earlier. Are the Customers, Accounts and Orders Regions/Subregions persistent or not? Not, since the regions were defined in native GemFire configuration (i.e. `cache.xml`) and will exist by the time the cache is initialized, or once the `<gfe:cache>` bean is created. Since the high-level region XML namespace abstractions, like `replicated-region`, perform the lookup first, it uses the regions as defined in the `cache.xml` configuration file.

5.5.6. Data Persistence

Regions can be made persistent. GemFire ensures that all the data you put into a region that is configured for persistence will be written to disk in a way that it can be recovered the next time you create the region. This allows data to be recovered after a machine or process failure or after an orderly shutdown and restart of GemFire.

To enable persistence with Spring Data GemFire, simply set the `persistent` attribute to true:

```
<gfe:partitioned-region id="persitent-partition" persistent="true"/>
```

IMPORTANT	Persistence for partitioned regions is supported from GemFire 6.5 onwards - configuring this option on a previous release will trigger an initialization exception.

Persistence may also be configured using the `data-policy` attribute, set to one of GemFire's data policy settings. For instance...

```
<gfe:partitioned-region id="persitent-partition" data-policy="PERSISTENT_PARTITION"/>
```

The data policy must match the region type and must also agree with the `persistent` attribute if explicitly set. An initialization exception will be thrown if, for instance, the `persistent` attribute is set to false, yet a persistent data policy was specified.

When persisting regions, it is recommended to configure the storage through the `disk-store` element for maximum efficiency. The diskstore is referenced using the disk-store-ref attribute. Additionally, the region may perform disk writes synchronously or asynchronously:

```
<gfe:partitioned-region id="persitent-partition" persistent="true" disk-store-ref=
"myDiskStore" disk-synchronous="true"/>
```

This is discussed further in Configuring a Disk Store

5.5.7. Subscription Interest Policy

GemFire allows configuration of subscriptions to control peer to peer event handling. Spring Data GemFire provides a `<gfe:subscription/>` to set the interest policy on replicated and partitioned regions to either `ALL` or `CACHE_CONTENT`.

```
<gfe:partitioned-region id="subscription-partition">
     <gfe:subscription type="CACHE_CONTENT"/>
</gfe:partitioned-region>
```

5.5.8. Data Eviction and Overflowing

Based on various constraints, each region can have an eviction policy in place for evicting data from memory. Currently, in GemFire, eviction applies to the least recently used entry (also known as LRU). Evicted entries are either destroyed or paged to disk (also known as **overflow**).

Spring Data GemFire supports all eviction policies (entry count, memory and heap usage) for both `partitioned-region` and `replicated-region` as well as `client-region`, through the nested `eviction` element. For example, to configure a partition to overflow to disk if its size is more then 512 MB, one could use the following configuration:

```
<gfe:partitioned-region id="overflow-partition">
     <gfe:eviction type="MEMORY_SIZE" threshold="512" action="OVERFLOW_TO_DISK"/>
</gfe:partitioned-region>
```

IMPORTANT	Replicas cannot use a `local destroy` eviction since that would invalidate them. See the GemFire docs for more information.

When configuring regions for overflow, it is recommended to configure the storage through the `disk-store` element for maximum efficiency.

For a detailed description of eviction policies, see the GemFire documentation (such as this page).

5.5.9. Data Expiration

GemFire allows you to control how long entries exist in the cache. Expiration is driven by elapsed time, as opposed to Eviction, which is driven by memory usage. Once an entry expires it may no longer be accessed from the cache.

GemFire supports the following Expiration types:

- **Time-to-Live (TTL)** - The amount of time, in seconds, the object may remain in the cache after the last creation or update. For entries, the counter is set to zero for create and put operations. Region counters are reset when the Region is created and when an entry has its counter reset.
- **Idle Timeout (TTI)** - The amount of time, in seconds, the object may remain in the cache after the last access. The Idle Timeout counter for an object is reset any time its TTL counter is reset. In addition, an entry's Idle Timeout counter is reset any time the entry is accessed through a get operation or a netSearch . The Idle Timeout counter for a Region is reset whenever the Idle Timeout is reset for one of its entries.

Each of these may be applied to the Region itself or entries in the Region. Spring Data GemFire provides `<region-ttl>`, `<region-tti>`, `<entry-ttl>` and `<entry-tti>` Region child elements to specify timeout values and expiration actions.

5.5.10. Annotation-based Data Expiration

As of Spring Data GemFire 1.7, a developer now has the ability to define Expiration policies and settings on individual Region Entry values, or rather, application domain objects directly. For instance, a developer might define Expiration settings on a Session-based application domain object like so...

```
@Expiration(timeout = "1800", action = "INVALIDATE")
public static class SessionBasedApplicationDomainObject {
}
```

In addition, a developer may also specify Expiration type specific settings on Region Entries using `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` for Idle Timeout (TTI) and Time-to-Live (TTL) Expiration, respectively...

```
@TimeToLiveExpiration(timeout = "3600", action = "LOCAL_DESTROY")
@IdleTimeoutExpiration(timeout = "1800", action = "LOCAL_INVALIDATE")
@Expiration(timeout = "1800", action = "INVALIDATE")
public static class AnotherSessionBasedApplicationDomainObject {
}
```

Both `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` take precedence over the generic `@Expiration` annotation when more than one Expiration annotation type is specified, as shown above. Though, neither `@IdleTimeoutExpiration` nor `@TimeToLiveExpiration` overrides the other; rather they may compliment each other when different Region Entry Expiration types, such as TTL and TTI, are configured.

NOTE

All @Expiration-based annotations apply only to Region Entry values. Expiration for a "Region" is not covered by Spring Data GemFire's Expiration annotation support. However, GemFire and Spring Data GemFire do allow you to set Region Expiration using the SDG XML namespace, like so...

```
<gfe:*-region id="Example" persistent="false">
  <gfe:region-ttl timeout="600" action="DESTROY"/>
  <gfe:region-tti timeout="300" action="INVALIDATE"/>
</gfe:*-region>
```

Spring Data GemFire's @Expiration annotation support is implemented with GemFire's `CustomExpiry` interface. See GemFire's User Guide for more details

The Spring Data GemFire `AnnotationBasedExpiration` class (and `CustomExpiry` implementation) is specifically responsible for processing the SDG @Expiration annotations and applying the Expiration policy and settings appropriately for Region Entry Expiration on request.

To use Spring Data GemFire to configure specific GemFire Regions to appropriately apply the Expiration policy and settings applied to your application domain objects annotated with @Expiration-based annotations, you must...

1. Define a Spring bean in the Spring ApplicationContext of type `AnnotationBasedExpiration` using the appropriate constructor or one of the convenient factory methods. When configuring Expiration for a specific Expiration type, such as Idle Timeout or Time-to-Live, then you should use one of the factory methods of the `AnnotationBasedExpiration` class, like so...

```
<bean id="ttlExpiration" class=
"org.springframework.data.gemfire.support.AnnotationBasedExpiration"
      factory-method="forTimeToLive"/>

<gfe:partitioned-region id="Example" persistent="false">
    <gfe:custom-entry-ttl ref="ttlExpiration"/>
</gfe:partitioned-region>
```

NOTE

To configure Idle Timeout (TTI) Expiration instead, then you would of course use the `forIdleTimeout` factory method along with the `<gfe:custom-entry-tti ref="ttiExpiration"/>` element to set TTI.

2. (optional) Annotate your application domain objects that will be stored in the Region with Expiration policies and custom settings using one of Spring Data GemFire's @Expiration annotations: `@Expiration`, `@IdleTimeoutExpiration` and/or `@TimeToLiveExpiration`
3. (optional) In cases where particular application domain objects have not been annotated with Spring Data GemFire's @Expiration annotations at all, but the GemFire Region is configured to use SDG's custom `AnnotationBasedExpiration` class to determine the Expiration policy and settings for objects stored in the Region, then it is possible to set "default" Expiration attributes on the `AnnotationBasedExpiration` bean by doing the following...

```
<bean id="defaultExpirationAttributes" class=
"com.gemstone.gemfire.cache.ExpirationAttributes">
    <constructor-arg value="600"/>
    <constructor-arg value="#{T(com.gemstone.gemfire.cache.ExpirationAction).DESTROY}
"/>
</bean>

<bean id="ttiExpiration" class=
"org.springframework.data.gemfire.support.AnnotationBasedExpiration"
      factory-method="forIdleTimeout">
    <constructor-arg ref="defaultExpirationAttributes"/>
</bean>

<gfe:partitioned-region id="Example" persistent="false">
    <gfe:custom-entry-tti ref="ttiExpiration"/>
</gfe:partitioned-region>
```

You may have noticed that the Spring Data GemFire's @Expiration annotations use String as the attributes type, rather than and perhaps more appropriately being strongly typed, i.e. `int` for 'timeout' and SDG'S `ExpirationActionType` for 'action'. Why is that?

Well, enter one of Spring Data GemFire's other features, leveraging Spring's core infrastructure for configuration convenience: Property Placeholders and Spring Expression Language (SpEL).

For instance, a developer can specify both the Expiration 'timeout' and 'action' using Property Placeholders in the @Expiration annotation attributes...

```
@TimeToLiveExpiration(timeout = "${gemfire.region.entry.expiration.ttl.timeout}"
    action = "${gemfire.region.entry.expiration.ttl.action}")
public class ExampleApplicationDomainObject {
}
```

Then, in your Spring context XML or in JavaConfig, you would declare the following beans...

```
<util:properties id="expirationSettings">
  <prop key="gemfire.region.entry.expiration.ttl.timeout">600</prop>
  <prop key="gemfire.region.entry.expiration.ttl.action">INVALIDATE</prop>
  ...
</util:properties>

<context:property-placeholder properties-ref="expirationProperties"/>
```

This is both convenient when multiple application domain objects might share similar Expiration policies and settings, or when you wish to externalize the configuration.

However, a developer may want more dynamic Expiration configuration determined by the state of the running system. This is where the power of SpEL comes in and is the recommended approach.

Not only can you refer to beans in the Spring context and access bean properties, invoke methods, etc, the values for Expiration 'timeout' and 'action' can be strongly typed. For example (building on the example above)...

```
<util:properties id="expirationSettings">
  <prop key="gemfire.region.entry.expiration.ttl.timeout">600</prop>
  <prop key="gemfire.region.entry.expiration.ttl.action"
>#{T(org.springframework.data.gemfire.ExpirationActionType).DESTROY}</prop>
  <prop key="gemfire.region.entry.expiration.tti.action"
>#{T(com.gemstone.gemfire.cache.ExpirationAction).INVALIDATE}</prop>
  ...
</util:properties>

<context:property-placeholder properties-ref="expirationProperties"/>
```

Then, on your application domain object...

```
@TimeToLiveExpiration(timeout =
"@expirationSettings['gemfire.region.entry.expiration.ttl.timeout']"
    action = "@expirationSetting['gemfire.region.entry.expiration.ttl.action']")
public class ExampleApplicationDomainObject {
}
```

You can imagine that the 'expirationSettings' bean could be a more interesting and useful object rather than a simple instance of `java.util.Properties`. In this example, even the Properties ('expirationSettings') using using SpEL to based the action value on the actual Expiration action enumerated types leading to more quickly identified failures if the types ever change.

All of this has been demonstrated and tested in the Spring Data GemFire test suite, by way of example. See the source for further details.

5.5.11. Local Region

Spring Data GemFire offers a dedicated `local-region` element for creating local regions. Local regions, as the name implies, are standalone meaning they do not share data with any other distributed system member. Other than that, all common region configuration options are supported. A minimal declaration looks as follows (again, the example relies on the Spring Data GemFire namespace naming conventions to wire the cache):

```
<gfe:local-region id="myLocalRegion" />
```

Here, a local region is created (if one doesn't exist already). The name of the region is the same as the bean id (myLocalRegion) and the bean assumes the existence of a GemFire cache named `gemfireCache`.

5.5.12. Replicated Region

One of the common region types is a **replicated region** or **replica**. In short, when a region is configured to be a replicated region, every member that hosts that region stores a copy of the region's entries locally. Any update to a replicated region is distributed to all copies of the region. When a replica is created, it goes through an initialization stage in which it discovers other replicas and automatically copies all the entries. While one replica is initializing you can still continue to use the other replica.

Spring Data GemFire offers a `replicated-region` element. A minimal declaration looks as follows. All common configuration options are available for replicated regions.

```
<gfe:replicated-region id="simpleReplica" />
```

5.5.13. Partitioned Region

Another region type supported out of the box by the Spring Data GemFire namespace is the partitioned region. To quote the GemFire docs:

"A partitioned region is a region where data is divided between peer servers hosting the region so that each peer stores a subset of the data. When using a partitioned region, applications are presented with a logical view of the region that looks like a single map containing all of the data in the region. Reads or writes to this map are transparently routed to the peer that hosts the entry that is the target of the operation. [...] GemFire divides the domain of hashcodes into buckets. Each bucket is assigned to a specific peer, but may be relocated at any time to another peer in order to improve the utilization of resources across the cluster."

A partition is created using the `partitioned-region` element. Its configuration options are similar to that of the `replicated-region` plus the partion specific features such as the number of redundant copies, total maximum memory, number of buckets, partition resolver and so on. Below is a quick example on setting up a partition region with 2 redundant copies:

```
<!-- bean definition named 'distributed-partition' backed by a region named
'redundant' with 2 copies
and a nested resolver declaration  -->
<gfe:partitioned-region id="distributed-partition" copies="2" total-buckets="4" name=
"redundant">
    <gfe:partition-resolver>
        <bean class="some.pkg.SimplePartitionResolver"/>
    </gfe:partition-resolver>
</gfe:partitioned-region>
```

`partitioned-region` Options

The following table offers a quick overview of configuration options specific to partitioned regions. These are in addition to the common region configuration options described above.

Table 3. partitioned-region options

Name	Values	Description
partition-resolver	**bean name**	The name of the partitioned resolver used by this region, for custom partitioning.
partition-listener	**bean name**	The name of the partitioned listener used by this region, for handling partition events.
copies	0..4	The number of copies for each partition for high-availability. By default, no copies are created meaning there is no redundancy. Each copy provides extra backup at the expense of extra storage.
colocated-with	**valid region name**	The name of the partitioned region with which this newly created partitioned region is colocated.
local-max-memory	**positive integer**	The maximum amount of memory, in megabytes, to be used by the region in **this** process.
total-max-memory	**any integer value**	The maximum amount of memory, in megabytes, to be used by the region in **all** processes.
recovery-delay	**any long value**	The delay in milliseconds that existing members will wait before satisfying redundancy after another member crashes. -1 (the default) indicates that redundancy will not be recovered after a failure.
startup-recovery-delay	**any long value**	The delay in milliseconds that new members will wait before satisfying redundancy. -1 indicates that adding new members will not trigger redundancy recovery. The default is to recover redundancy immediately when a new member is added.

5.5.14. Client Region

GemFire supports various deployment topologies for managing and distributing data. The topic is outside the scope of this documentation. However, to quickly recap, they can be classified in short as: peer-to-peer (p2p), client-server, and wide area network (or WAN). In the last two configurations, it is common to declare **client** regions which connect to a cache server. *Spring Data GemFire* offers dedicated support for such configuration through Configuring a GemFire ClientCache, `client-region` and `pool` elements. As the names imply, the former defines a client region while the latter defines connection pools to be used/shared by the various client regions.

Below is a typical client region configuration:

```
<!-- client region using the default client-cache pool -->
<gfe:client-region id="simple">
    <gfe:cache-listener ref="c-listener"/>
</gfe:client-region>

<!-- region using its own dedicated pool -->
<gfe:client-region id="complex" pool-name="gemfire-pool">
    <gfe:cache-listener ref="c-listener"/>
</gfe:client-region>

<bean id="c-listener" class="some.pkg.SimpleCacheListener"/>

<!-- pool declaration -->
<gfe:pool id="gemfire-pool" subscription-enabled="true">
    <gfe:locator host="someHost" port="40403"/>
</gfe:pool>
```

As with the other region types, `client-region` supports `CacheListener``s` as well as a single `CacheLoader` or `CacheWriter`. It also requires a connection `pool` for connecting to a server. Each client can have its own pool or they can share the same one.

NOTE

In the above example, the pool is configured with a `locator`. The locator is a separate process used to discover cache servers in the distributed system and are recommended for production systems. It is also possible to configure the pool to connect directly to one or more cache servers using the `server` element.

For a full list of options to set on the client and especially on the pool, please refer to the Spring Data GemFire schema (Spring Data GemFire Schema) and the GemFire documentation.

Client Interests

To minimize network traffic, each client can define its own 'interest', pointing out to GemFire, the data it actually needs. In Spring Data GemFire, interests can be defined for each client, both key-based and regular-expression-based types being supported; for example:

```
<gfe:client-region id="complex" pool-name="gemfire-pool">
    <gfe:key-interest durable="true" result-policy="KEYS">
        <bean id="key" class="java.lang.String">
             <constructor-arg value="someKey" />
        </bean>
    </gfe:key-interest>
    <gfe:regex-interest pattern=".*" receive-values="false"/>
</gfe:client-region>
```

A special key `ALL_KEYS` means interest is registered for all keys (identical to a regex interest of `.*`). The `receive-values` attribute indicates whether or not the values are received for create and update events. If true, values are received; if false, only invalidation events are received - refer to the

GemFire documentation for more details.

5.5.15. JSON Support

Gemfire 7.0 introduced support for caching JSON documents with OQL query support. These are stored internally as PdxInstance types using the JSONFormatter to perform conversion to and from JSON strings. Spring Data GemFire provides a `<gfe-data:json-region-autoproxy/>` tag to enable a AOP with Spring component to advise appropriate region operations, effectively encapsulating the JSONFormatter, allowing your application to work directly with JSON strings. In addition, Java objects written to JSON configured regions will be automatically converted to JSON using the Jackson ObjectMapper. Reading these values will return a JSON string.

By default, `<gfe-data:json-region-autoproxy/>` will perform the conversion on all regions. To apply this feature to selected regions, provide a comma delimited list of their ids via the `region-refs` attribute. Other attributes include a `pretty-print` flag (false by default) and `convert-returned-collections`. By default the results of region operations getAll() and values() will be converted for configured regions. This is done by creating a parallel structure in local memory. This can incur significant overhead for large collections. Set this flag to false to disable automatic conversion for these operation. NOTE: Certain region operations, specifically those that use GemFire's proprietary Region.Entry such as entries(boolean), entrySet(boolean) and getEntry() type are not targeted for AOP advice. In addition, the entrySet() method which returns a Set<java.util.Map.Entry<?,?>> is not affected.

```
<gfe-data:json-region-autoproxy pretty-print="true" region-refs="myJsonRegion"
convert-returned-collections="true"/>
```

This feature also works with seamlessly with GemfireTemplate operations, provided that the template is declared as a Spring bean. Currently native QueryService operations are not supported.

5.6. Creating an Index

GemFire allows the creation of indexes (or indices) to improve the performance of (common) queries. Spring Data GemFire allows indices to be declared through the `index` element:

```
<gfe:index id="myIndex" expression="someField" from="/someRegion"/>
```

Before creating an Index, Spring Data GemFire will verify whether an Index with the same name already exists. If an Index with the same name does exist, by default, SDG will "override" the existing Index by removing the old Index first followed by creating a new Index with the same name based on the new definition, regardless if the old definition was the same or not. To prevent the named Index definition change, especially when the old and new Index definitions may not match, set the `override` property to false, which effectively returns the existing Index definition by name.

Note that index declarations are not bound to a Region but rather are top-level elements (just like `gfe:cache`). This allows one to declare any number of indices on any Region whether they are just

created or already exist - an improvement over GemFire's native `cache.xml`. By default, the index relies on the default cache declaration but one can customize it accordingly or use a pool (if need be) - see the namespace schema for the full set of options.

5.7. Configuring a Disk Store

As of Release 1.2.0, Spring Data GemFire supports disk store configuration via a top level `disk-store` element.

NOTE	Prior to Release 1.2.0, `disk-store` was a child element of `*-region`. If you have regions configured with disk storage using a prior release of Spring Data GemFire and want to upgrade to the latest release, move the disk-store element to the top level, assign an id and use the region's `disk-store-ref` attribute. Also, `disk-synchronous` is now a region level attribute.

```
<gfe:disk-store id="diskStore1" queue-size="50" auto-compact="true"
        max-oplog-size="10" time-interval="9999">
        <gfe:disk-dir location="/gemfire/store1/" max-size="20"/>
        <gfe:disk-dir location="/gemfire/store2/" max-size="20"/>
</gfe:disk-store>
```

Disk stores are used by regions for file system persistent backup or overflow storage of evicted entries, and persistent backup of WAN gateways. Note that multiple components may share the same disk store. Also multiple directories may be defined for a single disk store. Please refer to the GemFire documentation for an explanation of the configuration options.

5.8. Using the GemFire Snapshot Service

Spring Data GemFire supports `Cache` and `Region` snapshots using GemFire's Snapshot Service. The out-of-the-box Snapshot Service support offers several convenient features to simply the use of GemFire's Cache and Region Snapshot Service APIs.

As GemFire documentation describes, snapshots allow you to save and subsequently reload the data later, which can be useful for moving data between environments, say from production to a staging or test environment in order to reproduce data-related issues in a controlled context. You can imagine combining Spring Data GemFire's Snapshot Service support with Spring's bean definition profiles to load snapshot data specific to the environment as necessary.

Spring Data GemFire's support for GemFire's Snapshot Service begins with the `<gfe-data:snapshot-service>` element from the GFE Data Access Namespace. For example, I might define Cache-wide snapshots to be loaded as well as saved with a couple snapshot imports and a single data export definition as follows:

```
<gfe-data:snapshot-service id="gemfireCacheSnapshotService">
  <gfe-data:snapshot-import location=
"/absolute/filesystem/path/to/import/fileOne.snapshot"/>
  <gfe-data:snapshot-import location=
"relative/filesystem/path/to/import/fileTwo.snapshot"/>
  <gfe-data:snapshot-export
      location="/absolute/or/relative/filesystem/path/to/export/directory"/>
</gfe-data:snapshot-service>
```

You can define as many imports and/or exports as you like. You can define just imports or just exports. The file locations and directory paths can be absolute, or relative to the Spring Data GemFire application JVM process's working directory.

This is a pretty simple example and the snapshot service defined in this case refers to the GemFire `Cache`, having a default name of `gemfireCache` (as described in Configuring a GemFire Cache). If you name your cache bean definition something different, than you can use the `cache-ref` attribute to refer to the cache bean by name:

```
<gfe:cache id="myCache"/>
...
<gfe-data:snapshot-service id="mySnapshotService" cache-ref="myCache">
...
</gfe-data:snapshot-service>
```

It is also straightforward to define a snapshot service for a GemFire Region by specifying the `region-ref` attribute:

```
<gfe:partitioned-region id="Example" persistent="false" .../>
...
<gfe-data:snapshot-service id="gemfireCacheRegionSnapshotService" region-ref="Example
">
  <gfe-data:snapshot-import location="relative/path/to/import/example.snapshot/>
  <gfe-data:snapshot-export location="/path/to/export/example.snapshot/>
</gfe-data:snapshot-service>
```

When the `region-ref` attribute is specified the Spring Data GemFire `SnapshotServiceFactoryBean` resolves the `region-ref` attribute to a Region bean defined in the Spring context and then proceeds to create a RegionSnapshotService. Again, the snapshot import and export definitions function the same way, however, the `location` must refer to a file on export.

NOTE	GemFire is strict about imported snapshot files actually existing before they are referenced. For exports, GemFire will create the snapshot file if it does not already exist. If the snapshot file for export already exists, the data will be overwritten.

NOTE	Spring Data GemFire includes a `suppress-import-on-init` attribute to the `<gfe-data:snapshot-service>` element to suppress the configured snapshot service from trying to import data into the Cache or a Region on initialization. This is useful when data exported from 1 Region is used to feed the import of another Region, for example.

5.8.1. Snapshot Location

For a `Cache`-based SnapshotService (i.e. a GemFire CacheSnapshotService) a developer would typically pass it a directory containing all the snapshot files to load rather than individual snapshot files, as the overloaded load method in the `CacheSnapshotService` API indicates.

NOTE	Of course, a developer may use the other, overloaded `load(:File[], :SnapshotFormat, :SnapshotOptions)` method variant to get specific about which snapshot files are to be loaded into the GemFire `Cache`.

However, Spring Data GemFire recognizes that a typical developer workflow might be to extract and export data from one environment into several snapshot files, zip all of them up, and then conveniently move the ZIP file to another environment for import.

As such, Spring Data GemFire enables the developer to specify a JAR or ZIP file on import for a `Cache`-based SnapshotService as follows:

```
<gfe-data:snapshot-service id="cacheBasedSnapshotService" cache-ref="gemfireCache">
  <gfe-data:snapshot-import location="/path/to/snapshots.zip"/>
</gfe-data:snapshot-service>
```

Spring Data GemFire will conveniently extract the provided ZIP file and treat it like a directory import (load).

5.8.2. Snapshot Filters

The real power of defining multiple snapshot imports and exports is realized through the use of snapshot filters. Snapshot filters implement GemFire's SnapshotFilter interface and are used to filter Region entries for inclusion into the Region on import and for inclusion into the snapshot on export.

Spring Data GemFire makes it brain dead simple to utilize snapshot filters on import and export using the `filter-ref` attribute or an anonymous, nested bean definition:

```
<gfe:cache/>

<gfe:partitioned-region id="Admins" persistent="false"/>
<gfe:partitioned-region id="Guests" persistent="false"/>

<bean id="activeUsersFilter" class=
"org.example.app.gemfire.snapshot.filter.ActiveUsersFilter/>

<gfe-data:snapshot-service id="adminsSnapshotService" region-ref="Admins">
  <gfe-data:snapshot-import location="/path/to/import/users.snapshot">
    <bean class="org.example.app.gemfire.snapshot.filter.AdminsFilter/>
  </gfe-data:snapshot-import>
  <gfe-data:snapshot-export location="/path/to/export/active/admins.snapshot"
                            filter-ref="activeUsersFilter"/>
</gfe-data:snapshot-service>

<gfe-data:snapshot-service id="guestsSnapshotService" region-ref="Guests">
  <gfe-data:snapshot-import location="/path/to/import/users.snapshot">
    <bean class="org.example.app.gemfire.snapshot.filter.GuestsFilter/>
  </gfe-data:snapshot-import>
  <gfe-data:snapshot-export location="/path/to/export/active/guests.snapshot"
                            filter-ref="activeUsersFilter"/>
</gfe-data:snapshot-service>
```

In addition, more complex snapshot filters can be expressed with the `ComposableSnapshotFilter` Spring Data GemFire class. This class implements GemFire's SnapshotFilter interface as well as the Composite software design pattern. In a nutshell, the Composite design pattern allows developers to compose multiple objects of the same type and treat the conglomerate as single instance of the object type, a very powerful and useful abstraction to be sure.

The `ComposableSnapshotFilter` has two factory methods, `'and'` and `'or'`, allowing developers to logically combine individual snapshot filters using the AND and OR logical operators, respectively. The factory methods just take a list of snapshot filters.

One is only limited by his/her imagination to leverage this powerful construct, for instance:

```
<bean id="activeUsersSinceFilter" class=
"org.springframework.data.gemfire.snapshot.filter.ComposableSnapshotFilter"
      factory-method="and">
  <constructor-arg index="0">
    <list>
      <bean class="org.example.app.gemfire.snapshot.filter.ActiveUsersFilter"/>
      <bean class="org.example.app.gemfire.snapshot.filter.UsersSinceFilter"
            p:since="2015-01-01"/>
    </list>
  </constructor-arg>
</bean>
```

You could then go onto combine the `activesUsersSinceFilter` with another filter using `'or'` like so:

```
<bean id="covertOrActiveUsersSinceFilter" class=
"org.springframework.data.gemfire.snapshot.filter.ComposableSnapshotFilter"
      factory-method="or">
  <constructor-arg index="0">
    <list>
      <ref bean="activeUsersSinceFilter"/>
      <bean class="org.example.app.gemfire.snapshot.filter.CovertUsersFilter"/>
    </list>
  </constructor-arg>
</bean>
```

5.8.3. Snapshot Events

By default, Spring Data GemFire uses GemFire's Snapshot Services on startup to import data and shutdown to export data. However, you may want to trigger periodic, event-based snapshots, for either import or export from within your application.

For this purpose, Spring Data GemFire defines two additional Spring application events (extending Spring's ApplicationEvent class) for imports and exports, respectively: `ImportSnapshotApplicationEvent` and `ExportSnapshotApplicationEvent`.

The two application events can be targeted at the entire GemFire Cache, or individual GemFire Regions. The constructors of these `ApplicationEvent` classes accept an optional Region pathname (e.g. "/Example") as well as 0 or more `SnapshotMetadata` instances.

The array of `SnapshotMetadata` is used to override the snapshot meta-data defined by `<gfe-data:snapshot-import>` and `<gfe-data:snapshot-export>` sub-elements in XML, which will be used in cases where snapshot application events do not explicitly provide `SnapshotMetadata`. Each individual `SnapshotMetadata` instance can define it's own `location` and `filters` properties.

Import/export snapshot application events are received by all snapshot service beans defined in the Spring application context. However, import/export events are only processed by "matching" snapshot service beans.

A Region-based `[Import|Export]SnapshotApplicationEvent` matches if the snapshot service bean defined is a `RegionSnapshotService` and it's Region reference (as determined by `region-ref`) matches the Region's pathname specified by the snapshot application event. A Cache-based `[Import|Export]SnapshotApplicationEvent` (i.e. a snapshot application event without a Region pathname) triggers all snapshot service beans, including any `RegionSnapshotService` beans, to perform either an import or export, respectively.

It is very easy to use Spring's ApplicationEventPublisher interface to fire import and/or export snapshot application events from your application like so:

```
@Component
public class ExampleApplicationComponent {

  @Autowired
  private ApplicationEventPublisher eventPublisher;

  @Resource(name = "Example")
  private Region<?, ?> example;

  public void someMethod() {
    ...

    SnapshotFilter myFilter = ...;

    SnapshotMetadata exportSnapshotMetadata = new SnapshotMetadata(new File(System
.getProperty("user.dir"),
      "/path/to/export/data.snapshot"), myFilter, null);

    eventPublisher.publishEvent(new ExportSnapshotApplicationEvent(this, example
.getFullPath(), exportSnapshotMetadata);

    ...
  }
}
```

In this particular example, only the "/Example" Region's SnapshotService bean will pick up and handle the export event, saving the filtered "/Example" Region's data to the "data.snapshot" file in a sub-direcrtory of the application's working directory.

Using Spring application events and messaging subsystem is a good way to keep your application loosely coupled. It is also not difficult to imagine that the snapshot application events could be fired on a periodic basis using Spring's Scheduling services.

5.9. Configuring GemFire's Function Service

As of Release 1.3.0, Spring Data GemFire provides annotation support for implementing and registering functions. Spring Data GemFire also provides namespace support for registering GemFire Functions for remote function execution. Please refer to the GemFire documentation for more information on the function execution framework. Functions are declared as Spring beans and must implement the `com.gemstone.gemfire.cache.execute.Function` interface or extend `com.gemstone.gemfire.cache.execute.FunctionAdapter`. The namespace uses a familiar pattern to declare functions:

```
<gfe:function-service>
  <gfe:function>
      <bean class="com.company.example.Function1"/>
      <ref bean="function2"/>
  </gfe:function>
</gfe:function-service>

<bean id="function2" class="com.company.example.Function2"/>
```

5.10. Configuring WAN Gateways

WAN gateways provide a way to synchronize GemFire distributed systems across geographic distributed areas. As of Release 1.2.0, Spring Data GemFire provides namespace support for configuring WAN gateways as illustrated in the following examples:

5.10.1. WAN Configuration in GemFire 7.0

GemFire 7.0 introduces new APIs for WAN configuration. While the original APIs provided in GemFire 6 are still supported, it is recommended that you use the new APIs if you are using GemFire 7.0. The Spring Data GemFire namespace supports either. In the example below, `GatewaySender`s are configured for a partitioned region by adding child elements to the region (`gateway-sender` and `gateway-sender-ref`). The `GatewaySender` may register `EventFilter`s and `TransportFilters`. Also shown below is an example configuration of an `AsyncEventQueue` which must also be wired into a region (not shown).

```
<gfe:partitioned-region id="region-inner-gateway-sender" >
    <gfe:gateway-sender
        remote-distributed-system-id="1">
            <gfe:event-filter>
                <bean class="org.springframework.data.gemfire.example.SomeEventFilter
"/>
            </gfe:event-filter>
            <gfe:transport-filter>
                <bean class=
"org.springframework.data.gemfire.example.SomeTransportFilter"/>
            </gfe:transport-filter>
    </gfe:gateway-sender>
    <gfe:gateway-sender-ref bean="gateway-sender"/>
</gfe:partitioned-region>

<gfe:async-event-queue id="async-event-queue" batch-size="10" persistent="true" disk-
store-ref="diskstore"
         maximum-queue-memory="50">
    <gfe:async-event-listener>
        <bean class="org.springframework.data.gemfire.example.SomeAsyncEventListener
"/>
    </gfe:async-event-listener>
</gfe:async-event-queue>


<gfe:gateway-sender id="gateway-sender" remote-distributed-system-id="2">
    <gfe:event-filter>
         <ref bean="event-filter"/>
         <bean class="org.springframework.data.gemfire.example.SomeEventFilter"/>
    </gfe:event-filter>
    <gfe:transport-filter>
         <ref bean="transport-filter"/>
         <bean class="org.springframework.data.gemfire.example.SomeTransportFilter"/>
     </gfe:transport-filter>
</gfe:gateway-sender>

<bean id="event-filter" class=
"org.springframework.data.gemfire.example.AnotherEventFilter"/>
<bean id="transport-filter" class=
"org.springframework.data.gemfire.example.AnotherTransportFilter"/>
```

On the other end of a `GatewaySender` is a corresponding `GatewayReceiver` to receive gateway events. The `GatewayReceiver` may also be configured with `EventFilter`s and `TransportFilter`s.

```
<gfe:gateway-receiver id="gateway-receiver"
    start-port="12345" end-port="23456" bind-address="192.168.0.1">
        <gfe:transport-filter>
            <bean class="org.springframework.data.gemfire.example.SomeTransportFilter
"/>
        </gfe:transport-filter>
</gfe:gateway-receiver>
```

Please refer to the GemFire product document for a detailed explanation of all the configuration options.

5.10.2. WAN Configuration in GemFire 6.6

```
<gfe:cache/>

<gfe:replicated-region id="region-with-gateway" enable-gateway="true" hub-id="gateway-
hub"/>

<gfe:gateway-hub id="gateway-hub" manual-start="true">
    <gfe:gateway gateway-id="gateway">
         <gfe:gateway-listener>
             <bean class="com.company.example.MyGatewayListener"/>
         </gfe:gateway-listener>
         <gfe:gateway-queue maximum-queue-memory="5" batch-size="3"
                batch-time-interval="10" />
     </gfe:gateway>

     <gfe:gateway gateway-id="gateway2">
         <gfe:gateway-endpoint port="1234" host="host1" endpoint-id="endpoint1"/>
         <gfe:gateway-endpoint port="2345" host="host2" endpoint-id="endpoint2"/>
    </gfe:gateway>
</gfe:gateway-hub>
```

A region may synchronize all or part of its contents to a gateway hub used to access one or more remote systems. The region must set `enable-gateway` to `true` and specify the `hub-id`.

NOTE	If just a hub-id is specified, Spring Data GemFire automatically assumes that the gateway should be enabled.

Please refer to the GemFire product document for a detailed explanation of all the configuration options.

Chapter 6. Working with the GemFire APIs

Once the GemFire Cache and Regions have been configured they can be injected and used inside application objects. This chapter describes the integration with Spring's Transaction Management functionality and `DaoException` hierarchy. It also covers support for dependency injection of GemFire managed objects.

6.1. Exception Translation

Using a new data access technology requires not only accommodating a new API but also handling exceptions specific to that technology. To accommodate this case, Spring Framework provides a technology agnostic, consistent exception hierarchy that abstracts the application from proprietary (and usually checked) exceptions to a set of focused runtime exceptions. As mentioned in the Spring Framework documentation, exception translation can be applied transparently to your data access objects through the use of the `@Repository` annotation and AOP by defining a `PersistenceExceptionTranslationPostProcessor` bean. The same exception translation functionality is enabled when using GemFire as long as at least a `CacheFactoryBean` is declared, e.g. using a `<gfe:cache/>` declaration, as it acts as an exception translator which is automatically detected by the Spring infrastructure and used accordingly.

6.2. GemfireTemplate

As with many other high-level abstractions provided by the Spring projects, Spring Data GemFire provides a **template** that simplifies GemFire data access. The class provides several **one-line** methods, for common region operations but also the ability to **execute** code against the native GemFire API without having to deal with GemFire checked exceptions for example through the `GemfireCallback`.

The template class requires a GemFire `Region` instance and once configured is thread-safe and should be reused across multiple classes:

```
<bean id="gemfireTemplate" class="org.springframework.data.gemfire.GemfireTemplate"
p:region-ref="someRegion"/>
```

Once the template is configured, one can use it alongside `GemfireCallback` to work directly with the GemFire `Region`, without having to deal with checked exceptions, threading or resource management concerns:

```
template.execute(new GemfireCallback<Iterable<String>>() {
    public Iterable<String> doInGemfire(Region reg) throws GemFireCheckedException,
GemFireException {
        // working against a Region of String
        Region<String, String> region = reg;

        region.put("1", "one");
        region.put("3", "three");

        return region.query("length < 5");
    }
});
```

For accessing the full power of the GemFire query language, one can use the `find` and `findUnique` which, as opposed to the `query` method, can execute queries across multiple regions, execute projections, and the like. The `find` method should be used when the query selects multiple items (through `SelectResults`) and the latter, `findUnique`, as the name suggests, when only one object is returned.

6.3. Support for Spring Cache Abstraction

Since 1.1, Spring Data GemFire provides an implementation of the Spring 3.1 cache abstraction. To use GemFire as a backing implementation, simply add `GemfireCacheManager` to your configuration:

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns:gfe="http://www.springframework.org/schema/gemfire"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
                      http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
                      http://www.springframework.org/schema/cache
http://www.springframework.org/schema/cache/spring-cache.xsd">

  <!-- turn on declarative caching -->
  <cache:annotation-driven/>

  <gfe:cache id="gemfire-cache"/>

  <!-- declare GemFire Cache Manager -->
  <bean id="cacheManager" class=
"org.springframework.data.gemfire.support.GemfireCacheManager" p:cache-ref="gemfire-
cache">
</beans>
```

6.4. Local, Cache Transaction Management

One of the most popular features of the *Spring Framework* is Transaction Management.

If you are not familiar with *Spring's* transaction abstraction then we strongly recommend reading about *Spring's Transaction Management* infrastructure as it offers a consistent *programming model* that works transparently across multiple APIs and can be configured either programmatically or declaratively (the most popular choice).

For GemFire, *Spring Data GemFire* provides a dedicated, per-cache, transaction manager that, once declared, allows Region operations to be executed atomically through Spring:

```
<gfe:transaction-manager id="tx-manager" cache-ref="cache"/>
```

NOTE

The example above can be simplified even further by eliminating the `cache-ref` attribute if the GemFire cache is defined under the default name, `gemfireCache`. As with the other *Spring Data GemFire* namespace elements, if the cache bean name is not configured, the aforementioned naming convention will be used. Additionally, the transaction manager name is "gemfireTransactionManager" if not explicitly specified.

Currently, Pivotal GemFire supports optimistic transactions with **read committed** isolation. Furthermore, to guarantee this isolation, developers should avoid making **in-place** changes that manually modify values present in the cache. To prevent this from happening, the transaction manager configures the cache to use **copy on read** semantics by default, meaning a clone of the actual value is created each time a read is performed. This behavior can be disabled if needed through the `copyOnRead` property.

For more information on the semantics and behavior of the underlying Geode transaction manager, please refer to the Geode CacheTransactionManager Javadoc as well as the documentation.

6.5. Global, JTA Transaction Management

It is also possible for Pivotal GemFire to participate in a Global, JTA based transaction, such as a transaction managed by an Java EE Application Server (e.g. WebSphere Application Server, a.k.a. WAS) using Container Managed Transactions (CMT) along with other JTA resources.

However, unlike many other JTA "compliant" resources (e.g. JMS Message Brokers like ActiveMQ), Pivotal GemFire is **not** an XA compliant resource. Therefore, Pivotal GemFire must be positioned as the "*Last Resource*" in a JTA transaction (*prepare phase*) since it does not implement the 2-phase commit protocol, or rather does not handle distributed transactions.

Many managed environments with CMT maintain support for "*Last Resource*", non-XA compliant resources in JTA transactions though it is not actually required in the JTA spec. More information on what a non-XA compliant, "*Last Resource*" means can be found in Red Hat's documentation. In fact, Red Hat's JBoss project, Narayana is one such LGPL Open Source implementation. *Narayana* refers to this as "*Last Resource Commit Optimization*" (LRCO). More details can be found here.

However, whether you are using Pivotal GemFire in a standalone environment with an Open Source JTA Transaction Management implementation that supports "*Last Resource*", or a managed environment (e.g. Java EE AS such as WAS), *Spring Data Geode* has you covered.

There are a series of steps you must complete to properly use Pivotal GemFire as a "*Last Resource*" in a JTA transaction involving more than 1 transactional resource. Additionally, there can only be 1 non-XA compliant resource (e.g. Pivotal GemFire) in such an arrangement.

1) First, you must complete Steps 1-4 in GemFire's documentation here.

NOTE	#1 above is independent of your *Spring [Boot] and/or [Data GemFire]* application and must be completed successfully.

2) Referring to Step 5 in GemFire's documentation, *Spring Data GemFire's* Annotation support will attempt to set the `GemFireCache`, `copyOnRead` property for you when using the `@EnableGemFireAsLastResource` annotation.

However, if SDG's auto-configuration is unsuccessful then you must explicitly set the `copy-on-read` attribute on the `<gfe:cache>` or `<gfe:client-cache>` element in XML or the `copyOnRead` property of the SDG `CacheFactoryBean` class in JavaConfig to **true**. For example...

Peer Cache XML:

```
<gfe:cache ... copy-on-read="true"/>
```

Peer Cache JavaConfig:

```
@Bean
CacheFacatoryBean gemfireCache() {

  CacheFactoryBean gemfireCache = new CacheFactoryBean();

  gemfireCache.setClose(true);
  gemfireCache.setCopyOnRead(true);

  return gemfireCache;
}
```

Client Cache XML:

```
<gfe:client-cache ... copy-on-read="true"/>
```

Client Cache JavaConfig:

```
@Bean
ClientCacheFacatoryBean gemfireCache() {

  ClientCacheFactoryBean gemfireCache = new ClientCacheFactoryBean();

  gemfireCache.setClose(true);
  gemfireCache.setCopyOnRead(true);

  return gemfireCache;
}
```

NOTE explicitly setting the `copy-on-read` attribute or optionally the `copyOnRead` property really should not be necessary.

3) At this point, you **skip** Steps 6-8 in GemFire's documentation and let *Spring Data Geode* work its magic. All you need do is annotate your *Spring* `@Configuration` class with *Spring Data GemFire's* **new** `@EnableGemFireAsLastResource` annotation and a combination of *Spring's* Transaction Management infrastructure and *Spring Data GemFire's* `@EnableGemFireAsLastResource` configuration does the trick.

The configuration looks like this...

```
@Configuration
@EnableGemFireAsLastResource
@EnableTransactionManagement(order = 1)
class GeodeConfiguration {

  ...
}
```

The only requirements are...

3.1) The `@EnableGemFireAsLastResource` annotation must be declared on the same *Spring* `@Configuration` class where *Spring's* `@EnableTransactionManagement` annotation is also specified.

3.2) The `order` attribute of the `@EnableTransactionManagement` annotation must be explicitly set to an integer value that is not `Integer.MAX_VALUE` or `Integer.MIN_VALUE` (defaults to `Integer.MAX_VALUE`).

Of course, hopefully you are aware that you also need to configure *Spring's* `JtaTransactionManager` when using JTA Transactions like so..

```
@Bean
public JtaTransactionManager transactionManager(UserTransaction userTransaction) {

   JtaTransactionManager transactionManager = new JtaTransactionManager();

   transactionManager.setUserTransaction(userTransaction);

   return transactionManager;
}
```

NOTE	The configuration in section Local, Cache Transaction Management does **not** apply here. The use of *Spring Data GemFire's* `GemfireTransactionManager` is applicable only in "Local", Cache Transactions, **not** "Global", JTA Transactions. Therefore, you do **not** configure the SDG `GemfireTransactionManager` in this case. You configure *Spring's* `JtaTransactionManager` as shown above.

For more details on using *Spring's Transaction Management* with JTA, see here.

Effectively, *Spring Data GemFire's* `@EnableGemFireAsLastResource` annotation imports configuration containing 2 Aspect bean definitions that handles the GemFire `o.a.g.ra.GFConnectionFactory.getConnection()` and `o.a.g.ra.GFConnection.close()` operations at the appropriate points during the transactional operation.

Specifically, the correct sequence of events are…

1. `jtaTransation.begin()`
2. `GFConnectionFactory.getConnection()`
3. Call the application's `@Transactional` service method
4. Either `jtaTransaction.commit()` or `jtaTransaction.rollback()`
5. Finally, `GFConnection.close()`

This is consistent with how you, as the application developer, would code this manually if you had to use the JTA API + GemFire API yourself, as shown in the GemFire example.

Thankfully, *Spring* does the heavy lifting for you and all you need do after applying the appropriate configuration (shown above) is…

```
@Service
class MyTransactionalService ... {

  @Transactional
  public <Return-Type> someTransactionalMethod() {
    // perform business logic interacting with and accessing multiple JTA resources
atomically, here
  }

  ...
}
```

#1 & #4 above are appropriately handled for you by *Spring's* JTA based `PlatformTransactionManager` once the `@Transactional` boundary is entered by your application (i.e. when the `MyTransactionSerivce.someTransactionalMethod()` is called).

#2 & #3 are handled by *Spring Data GemFire's* new Aspects enabled with the `@EnableGemFireAsLastResource` annotation.

#3 of course is the responsibility of your application.

Indeed, with the appropriate logging configured, you will see the correct sequence of events...

```
2017-Jun-22 11:11:37 TRACE TransactionInterceptor - Getting transaction for
[example.app.service.MessageService.send]

2017-Jun-22 11:11:37 TRACE GemFireAsLastResourceConnectionAcquiringAspect - Acquiring
GemFire Connection
from GemFire JCA ResourceAdapter registered at [gfe/jca]

2017-Jun-22 11:11:37 TRACE MessageService - PRODUCER [ Message :
[{ @type = example.app.domain.Message, id= MSG0000000000, message = SENT }],
JSON : [{"id":"MSG0000000000","message":"SENT"}] ]

2017-Jun-22 11:11:37 TRACE TransactionInterceptor - Completing transaction for
[example.app.service.MessageService.send]

2017-Jun-22 11:11:37 TRACE GemFireAsLastResourceConnectionClosingAspect - Closed
GemFire Connection @ [Reference [...]]
```

For more details on using Pivotal GemFire in JTA transactions, see here.

For more details on configuring Pivotal GemFire as a "*Last Resource*", see here.

6.6. GemFire Continuous Query Container

A powerful functionality offered by GemFire is continuous querying (or CQ). In short, CQ allows one to create a query and automatically be notified when new data that gets added to GemFire

matches the query. Spring GemFire provides dedicated support for CQs through the `org.springframework.data.gemfire.listener` package and its **listener container**; very similar in functionality and naming to the JMS integration in Spring Framework; in fact, users familiar with the JMS support in Spring, should feel right at home. Basically Spring Data GemFire allows methods on POJOs to become end-points for CQ - simply define the query and indicate the method that should be notified when there is a match - Spring Data GemFire takes care of the rest. This is similar Java EE's message-driven bean style, but without any requirement for base class or interface implementations, based on GemFire.

NOTE	Currently, continuous queries are supported by GemFire only in client/server topologies. Additionally the pool used is required to have the `subscription` property enabled. Please refer to the documentation for more information.

6.6.1. Continuous Query Listener Container

Spring Data GemFire simplifies the creation, registration, life-cycle and dispatch of CQs by taking care of the infrastructure around them through `ContinuousQueryListenerContainer` which does all the heavy lifting on behalf of the user - users familiar with EJB and JMS should find the concepts familiar as it is designed as close as possible to the support in Spring Framework and its message-driven POJOs (MDPs)

`ContinuousQueryListenerContainer` acts as an event (or message) listener container; it is used to receive the events from the registered CQs and drive the POJOs that are injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. It acts as the intermediary between an EDP (Event Driven POJO) and the event provider and takes care of creation and registration of CQs (to receive events), resource acquisition and release, exception conversion and the like. This allows you as an application developer to write the (possibly complex) business logic associated with receiving an event (and reacting to it), and delegates boilerplate GemFire infrastructure concerns to the framework.

The container is fully customizable - one can chose either to use the CQ thread to perform the dispatch (synchronous delivery) or a new thread (from an existing pool for examples) for an asynchronous approach by defining the suitable `java.util.concurrent.Executor` (or Spring's `TaskExecutor`). Depending on the load, the number of listeners or the runtime environment, one should change or tweak the executor to better serve her needs - in particular in managed environments (such as app servers), it is highly recommended to pick a a proper `TaskExecutor` to take advantage of its runtime.

6.6.2. The `ContinuousQueryListenerAdapter` and `ContinuousQueryListener`

The `ContinuousQueryListenerAdapter` class is the final component in Spring Data GemFire CQ support: in a nutshell, it allows you to expose almost **any** class as a EDP (there are of course some constraints) - it implements `ContinuousQueryListener`, a simpler listener interface similar to GemFire CqListener.

Consider the following interface definition. Notice the various event handling methods and their parameters:

```
public interface EventDelegate {
     void handleEvent(CqEvent event);
     void handleEvent(Operation baseOp);
     void handleEvent(Object key);
     void handleEvent(Object key, Object newValue);
     void handleEvent(Throwable th);
     void handleQuery(CqQuery cq);
     void handleEvent(CqEvent event, Operation baseOp, byte[] deltaValue);
     void handleEvent(CqEvent event, Operation baseOp, Operation queryOp, Object key,
Object newValue);
}
```

```
public class DefaultEventDelegate implements EventDelegate {
    // implementation elided for clarity...
}
```

In particular, note how the above implementation of the `EventDelegate` interface (the above `DefaultEventDelegate` class) has **no** GemFire dependencies at all. It truly is a POJO that we will make into an EDP via the following configuration (note that the class doesn't have to implement an interface, one is present only to better show case the decoupling between contract and implementation).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:gfe="http://www.springframework.org/schema/gemfire"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/gemfire
http://www.springframework.org/schema/gemfire/spring-gemfire.xsd">

    <gfe:client-cache pool-name="client"/>

    <gfe:pool id="client" subscription-enabled="true">
       <gfe:server host="localhost" port="40404"/>
    </gfe:pool>

    <gfe:cq-listener-container>
       <!-- default handle method -->
       <gfe:listener ref="listener" query="SELECT * from /region"/ >
       <gfe:listener ref="another-listener" query="SELECT * from /another-region"
name="my-query" method="handleQuery"/>
    </gfe:cq-listener-container>

    <bean id="listener" class="gemfireexample.DefaultMessageDelegate"/>
    <bean id="another-listener" class="gemfireexample.DefaultMessageDelegate"/>
  ...
<beans>
```

NOTE

The example above shows some of the various forms that a listener can have; at its minimum the listener reference and the actual query definition are required. It's possible however to specify a name for the resulting continuous query (useful for monitoring) but also the name of the method (the default is `handleEvent`). The specified method can have various argument types, the `EventDelegate` interface lists the allowed types.

The example above uses the Spring Data GemFire namespace to declare the event listener container and automatically register the listeners. The full blown, **beans** definition is displayed below:

```
<!-- this is the Event Driven POJO (MDP) -->
<bean id="eventListener" class=
"org.springframework.data.gemfire.listener.adapter.ContinuousQueryListenerAdapter">
    <constructor-arg>
        <bean class="gemfireexample.DefaultEventDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the event listener container... -->
<bean id="gemfireListenerContainer" class=
"org.springframework.data.gemfire.listener.ContinuousQueryListenerContainer">
    <property name="cache" ref="gemfireCache"/>
    <property name="queryListeners">
      <!-- set of listeners -->
      <set>
        <bean class=
"org.springframework.data.gemfire.listener.ContinuousQueryDefinition" >
                <constructor-arg value="SELECT * from /region" />
                <constructor-arg ref="eventListener" />
        </bean>
      </set>
    </property>
</bean>
```

Each time an event is received, the adapter automatically performs type translation between the GemFire event and the required method argument(s) transparently. Any exception caused by the method invocation is caught and handled by the container (by default, being logged).

6.7. Wiring `Declarable` components

GemFire XML configuration (usually named `cache.xml` allows **user** objects to be declared as part of the configuration. Usually these objects are `CacheLoader`s or other pluggable callback components supported by GemFire. Using native GemFire configuration, each user type declared through XML must implement the `Declarable` interface which allows arbitrary parameters to be passed to the declared class through a `Properties` instance.

In this section we describe how you can configure these pluggable components defined in `cache.xml` using Spring while keeping your Cache/Region configuration defined in `cache.xml` This allows your pluggable components to focus on the application logic and not the location or creation of DataSources or other collaboration objects.

However, if you are starting a green field project, it is recommended that you configure Cache, Region, and other pluggable components directly in Spring. This avoids inheriting from the `Declarable` interface or the base class presented in this section. See the following sidebar for more information on this approach.

Eliminate `Declarable` components

One can configure custom types entirely through Spring as mentioned in Configuring a GemFire Region. That way, one does not have to implement the `Declarable` interface and also benefits from all the features of the Spring IoC container (not just dependency injection but also life-cycle and instance management).

As an example of configuring a `Declarable` component using Spring, consider the following declaration (taken from the `Declarable` javadoc):

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <parameter name="URL">
     <string>jdbc://12.34.56.78/mydb</string>
   </parameter>
</cache-loader>
```

To simplify the task of parsing, converting the parameters and initializing the object, Spring Data GemFire offers a base class (`WiringDeclarableSupport`) that allows GemFire user objects to be wired through a **template** bean definition or, in case that is missing, perform autowiring through the Spring container. To take advantage of this feature, the user objects need to extend `WiringDeclarableSupport` which automatically locates the declaring `BeanFactory` and performs wiring as part of the initialization process.

Why is a base class needed?

In the current GemFire release there is no concept of an **object factory** and the types declared are instantiated and used as is. In other words, there is no easy way to manage object creation outside GemFire.

6.7.1. Configuration using template definitions

When used, `WiringDeclarableSupport` tries to first locate an existing bean definition and use that as wiring template. Unless specified, the component class name will be used as an implicit bean definition name. Let's see how our `DBLoader` declaration would look in that case:

```
public class DBLoader extends WiringDeclarableSupport implements CacheLoader {
  private DataSource dataSource;

  public void setDataSource(DataSource ds){
    this.dataSource = ds;
  }

  public Object load(LoaderHelper helper) { ... }
}
```

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- no parameter is passed (use the bean implicit name
   that is the class name) -->
</cache-loader>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="dataSource" ... />

     <!-- template bean definition -->
     <bean id="com.company.app.DBLoader" abstract="true" p:dataSource-ref="dataSource
"/>
</beans>
```

In the scenario above, as no parameter was specified, a bean with the id/name `com.company.app.DBLoader` was used as a template for wiring the instance created by GemFire. For cases where the bean name uses a different convention, one can pass in the `bean-name` parameter in the GemFire configuration:

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- pass the bean definition template name
        as parameter -->
   <parameter name="bean-name">
     <string>template-bean</string>
   </parameter>
</cache-loader>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="dataSource" ... />

     <!-- template bean definition -->
     <bean id="template-bean" abstract="true" p:dataSource-ref="dataSource"/>

</beans>
```

NOTE	The **template** bean definitions do not have to be declared in XML - any format is allowed (Groovy, annotations, etc..).

6.7.2. Configuration using auto-wiring and annotations

If no bean definition is found, by default, `WiringDeclarableSupport` will autowire the declaring instance. This means that unless any dependency injection **metadata** is offered by the instance, the container will find the object setters and try to automatically satisfy these dependencies. However, one can also use JDK 5 annotations to provide additional information to the auto-wiring process. We strongly recommend reading the dedicated chapter in the Spring documentation for more information on the supported annotations and enabling factors.

For example, the hypothetical `DBLoader` declaration above can be injected with a Spring-configured `DataSource` in the following way:

```
public class DBLoader extends WiringDeclarableSupport implements CacheLoader {
  // use annotations to 'mark' the needed dependencies
  @javax.inject.Inject
  private DataSource dataSource;

  public Object load(LoaderHelper helper) { ... }
}
```

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- no need to declare any parameters anymore
        since the class is auto-wired -->
</cache-loader>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd
           http://www.springframework.org/schema/context
           http://www.springframework.org/schema/context/spring-context.xsd">

     <!-- enable annotation processing -->
     <context:annotation-config/>

</beans>
```

By using the JSR-330 annotations, the cache loader code has been simplified since the location and creation of the DataSource has been externalized and the user code is concerned only with the loading process. The `DataSource` might be transactional, created lazily, shared between multiple objects or retrieved from JNDI - these aspects can be easily configured and changed through the Spring container without touching the `DBLoader` code.

Chapter 7. Working with GemFire Serialization

To improve overall performance of the data grid, GemFire supports a dedicated serialization protocol (PDX) that is both faster and offers more compact results over the standard Java serialization and works transparently across various language platforms (such as Java, .NET and C++). This chapter discusses the various ways in which Spring Data GemFire simplifies and improves GemFire custom serialization in Java.

7.1. Wiring deserialized instances

It is fairly common for serialized objects to have transient data. Transient data is often dependent on the node or environment where it lives at a certain point in time, for example a DataSource. Serializing such information is useless (and potentially even dangerous) since it is local to a certain VM/machine. For such cases, Spring Data GemFire offers a special `Instantiator` that performs wiring for each new instance created by GemFire during deserialization.

Through such a mechanism, one can rely on the Spring container to inject (and manage) certain dependencies making it easy to split transient from persistent data and have **rich domain objects** in a transparent manner (Spring users might find this approach similar to that of `@Configurable`). The `WiringInstantiator` works just like `WiringDeclarableSupport`, trying to first locate a bean definition as a wiring template and following to autowiring otherwise. Please refer to the previous section (Wiring `Declarable` components) for more details on wiring functionality.

To use this `Instantiator`, simply declare it as a usual bean:

```
<bean id="instantiator" class=
"org.springframework.data.gemfire.serialization.WiringInstantiator">
  <!-- DataSerializable type -->
  <constructor-arg>org.pkg.SomeDataSerializableClass</constructor-arg>
  <!-- type id -->
  <constructor-arg>95</constructor-arg>
</bean>
```

During the container startup, once it is being initialized, the `instantiator` will, by default, register itself with the GemFire system and perform wiring on all instances of `SomeDataSerializableClass` created by GemFire during deserialization.

7.2. Auto-generating custom `Instantiator`s

For data intensive applications, a large number of instances might be created on each machine as data flows in. Out of the box, GemFire uses reflection to create new types but for some scenarios, this might prove to be expensive. As always, it is good to perform profiling to quantify whether this is the case or not. For such cases, Spring Data GemFire allows the automatic generation of `Instatiator` classes which instantiate a new type (using the default constructor) without the use of reflection:

```
<bean id="instantiator-factory" class=
"org.springframework.data.gemfire.serialization.InstantiatorFactoryBean">
  <property name="customTypes">
    <map>
      <entry key="org.pkg.CustomTypeA" value="1025"/>
      <entry key="org.pkg.CustomTypeB" value="1026"/>
    </map>
  </property>
</bean>
```

The definition above, automatically generated two `Instantiator`s for two classes, namely `CustomTypeA` and `CustomTypeB` and registers them with GemFire, under user id `1025` and `1026`. The two instantiators avoid the use of reflection and create the instances directly through Java code.

Chapter 8. POJO mapping

8.1. Entity Mapping

Spring Data GemFire provides support to map entities that will be stored in a GemFire data grid. The mapping metadata is defined using annotations at the domain classes just like this:

Example 1. Mapping a domain class to a GemFire Region

```
@Region("People")
public class Person {

  @Id Long id;
  String firstname;
  String lastname;

  @PersistenceConstructor
  public Person(String firstname, String lastname) {
    // ···
  }

  ···
}
```

The first thing you see here is the `@Region` annotation that can be used to customize the Region in which the `Person` class is stored in. The `@Id` annotation can be used to annotate the property that shall be used as the Cache key. The `@PersistenceConstructor` annotation actually helps disambiguating multiple potentially available constructors taking parameters and explicitly marking the one annotated as the one to be used to create entities. With none or only a single constructor you can omit the annotation.

In addition to storing entities in top-level Regions, entities can be stored in GemFire Sub-Regions, as so:

```
@Region("/Users/Admin")
public class Admin extends User {
  ···
}

@Region("/Users/Guest")
public class Guest extends User {
  ···
}
```

Be sure to use the full-path of the GemFire Region, as defined in Spring Data GemFire XML namespace configuration meta-data, as specified in the `id` or `name` attributes of the `<*-region>` bean

definition.

As alternative to specifying the Region in which the entity will be stored using the `@Region` annotation on the entity class, you can also specify the `@Region` annotation on the entity's `Repository` abstraction. See GemFire Repositories for more details.

However, let's say you want to store a Person in multiple GemFire Regions (e.g. `People` and `Customers`), then you can define your corresponding `Repository` interface abstractions like so:

```
@Region("People")
public interface PersonRepository extends GemfireRepository<Person, String> {
...
}

@Region("Customers")
public interface CustomerRepository extends GemfireRepository<Person, String> {
...
}
```

8.2. Mapping PDX Serializer

Spring Data GemFire provides a custom `PDXSerializer` implementation that uses the mapping information to customize entity serialization. Beyond that it allows customizing the entity instantiation by using the Spring Data `EntityInstantiator` abstraction. By default the serializer uses a `ReflectionEntityInstantiator` that will use the persistence constructor of the mapped entity (either the single declared one or explicitly annoted with `@PersistenceConstructor`). To provide values for constructor parameters it will read fields with name of the constructor parameters from the `PDXReader` supplied.

Example 2. Using @Value on entity constructor parameters

```
public class Person {

  public Person(@Value("#root.foo") String firstname, @Value("bean") String
lastname) {
    // ···
  }

}
```

The entity annotated as such will get the field `foo` read from the `PDXReader` and handed as constructor parameter value for `firstname`. The value for `lastname` will be the Spring bean with name `bean`.

Chapter 9. GemFire Repositories

9.1. Introduction

Spring Data GemFire provides support to use the Spring Data Repository abstraction to easily persist entities into GemFire and execute queries. A general introduction to the Repository programming model has been provided here.

9.2. Spring Configuration

To bootstrap Spring Data Repositories you use the `<repositories/>` element from the GemFire Data namespace:

Example 3. Bootstrap GemFire Repositories

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:gfe-data="http://www.springframework.org/schema/data/gemfire"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-
beans.xsd
                           http://www.springframework.org/schema/data/gemfire

http://www.springframework.org/schema/data/gemfire/spring-data-gemfire.xsd>

  <gfe-data:repositories base-package="com.acme.repository" />

</beans>
```

This configuration snippet will look for interfaces below the configured base package and create Repository instances for those interfaces backed by a `SimpleGemFireRepository`. Note that you have to have your domain classes correctly mapped to configured Regions or the bootstrap process will fail otherwise.

9.3. Executing OQL Queries

The GemFire Repositories allow the definition of query methods to easily execute OQL Queries against the Region the managed entity is mapped to.

Example 4. Sample Repository

```
@Region("myRegion")
public class Person { ⋯ }
```

```
public interface PersonRepository extends CrudRepository<Person, Long> {

  Person findByEmailAddress(String emailAddress);

  Collection<Person> findByFirstname(String firstname);

  @Query("SELECT * FROM /Person p WHERE p.firstname = $1")
  Collection<Person> findByFirstnameAnnotated(String firstname);

  @Query("SELECT * FROM /Person p WHERE p.firstname IN SET $1")
  Collection<Person> findByFirstnamesAnnotated(Collection<String> firstnames);
}
```

The first method listed here will cause the following query to be derived: `SELECT x FROM /MyRegion x WHERE x.emailAddress = $1`. The second method works the same way except it's returning all entities found whereas the first one expects a single result value. In case the supported keywords are not sufficient to declare your query or the method name gets to verbose you can annotate the query methods with `@Query` as seen for methods 3 and 4.

Table 4. Supported keywords for query methods

Keyword	Sample	Logical result
`GreaterThan`	`findByAgeGreaterThan(int age)`	`x.age > $1`
`GreaterThanEqual`	`findByAgeGreaterThanEqual(int age)`	`x.age >= $1`
`LessThan`	`findByAgeLessThan(int age)`	`x.age < $1`
`LessThanEqual`	`findByAgeLessThanEqual(int age)`	`x.age <= $1`
`IsNotNull`, `NotNull`	`findByFirstnameNotNull()`	`x.firstname =! NULL`
`IsNull`, `Null`	`findByFirstnameNull()`	`x.firstname = NULL`
`In`	`findByFirstnameIn(Collection<String> x)`	`x.firstname IN SET $1`
`NotIn`	`findByFirstnameNotIn(Collection<String> x)`	`x.firstname NOT IN SET $1`
`IgnoreCase`	`findByFirstnameIgnoreCase(String firstName)`	`x.firstname.equalsIgnoreCase($1)`
(No keyword)	`findByFirstname(String name)`	`x.firstname = $1`
`Like`	`findByFirstnameLike(String name)`	`x.firstname LIKE $1`
`Not`	`findByFirstnameNot(String name)`	`x.firstname != $1`
`IsTrue`, `True`	`findByActiveIsTrue()`	`x.active = true`
`IsFalse`, `False`	`findByActiveIsFalse()`	`x.active = false`

9.4. OQL Query Extensions with Annotations

Many query languages, such as Pivotal GemFire's OQL (Object Query Language), have extensions that are not directly supported by the Spring Data Commons Repository infrastructure.

One of Spring Data Commons' Repository infrastructure goals is to function as the lowest common denominator to maintain support and portability across the widest array of data stores available and in use for application development today. Technically, this means developers can access multiple different data stores supported by Spring Data Commons within their applications by reusing their existing application-specific Repository interfaces, a very convenient and powerful abstraction.

To support GemFire's OQL Query language extensions and maintain portability across data stores, Spring Data GemFire adds support for OQL Query extensions by way of Java Annotations. These new Annotations will be ignored by other Spring Data Repository implementations (e.g. Spring Data Redis) that don't have similar query language extensions.

For instance, many data stores will most likely not implement GemFire's OQL `IMPORT` keyword. By implementing `IMPORT` as an Annotation (`@Import`) rather than as part of the query method signature (specifically, the method 'name'), this will not interfere with the parsing infrastructure when evaluating the query method name to construct the appropriate data store language appropriate query.

Currently, the set of OQL Query language extensions that are supported by Spring Data GemFire include:

Table 5. Supported OQL Query extensions for query methods

Keyword	Annotation	Description	Arguments
HINT	`@Hint`	OQL Query Index Hints	`String[]` (e.g. @Hint({ "IdIdx", "TxDateIdx" }))
IMPORT	`@Import`	Qualify application-specific types.	`String` (e.g. @Import("org.example.app.domain.Type"))
LIMIT	`@Limit`	Limit the returned query result set.	`Integer` (e.g. @Limit(10); default is Integer.MAX_VALUE)
TRACE	`@Trace`	Enable OQL Query specific debugging.	NA

As an example, suppose you have a `Customers` application domain type and corresponding GemFire Region along with a `CustomerRepository` and a query method to lookup `Customers` by last name, like so...

Example 5. Sample Repository

```
package ...;

import org.springframework.data.annotation.Id;
import org.springframework.data.gemfire.mapping.Region;
...

@Region("Customers")
public class Customer ... {

  @Id
  private Long id;

  ...
}
```

```
package ...;

import org.springframework.data.gemfire.repository.GemfireRepository;
...

public interface CustomerRepository extends GemfireRepository<Customer, Long> {

  @Trace
  @Limit(10)
  @Hint("LastNameIdx")
  @Import("org.example.app.domain.Customer")
  List<Customer> findByLastName(String lastName);

  ...
}
```

This will result in the following OQL Query:

```
<TRACE> <HINT 'LastNameIdx'> IMPORT org.example.app.domain.Customer; SELECT * FROM /Customers c
WHERE c.lastName = $1 LIMIT 10
```

Spring Data GemFire's Repository extension support is careful not to create conflicting declaratives when the Query Annotation extensions are used in combination with the `@Query` annotation.

For instance, suppose you have a raw `@Query` annotated query method defined in your `CustomerRepository` like so...

Example 6. CustomerRepository

```
public interface CustomerRepository extends GemfireRepository<Customer, Long> {

  @Trace
  @Limit(10)
  @Hint("CustomerIdx")
  @Import("org.example.app.domain.Customer")
  @Query("<TRACE> <HINT 'ReputationIdx'> SELECT DISTINCT * FROM /Customers c WHERE
c.reputation > $1 ORDER BY c.reputation DESC LIMIT 5")
  List<Customer>
findDistinctCustomersByReputationGreaterThanOrderByReputationDesc(Integer
reputation);
}
```

This query method results in the following OQL Query:

```
IMPORT org.example.app.domain.Customer; <TRACE> <HINT 'ReputationIdx'> SELECT DISTINCT * FROM
/Customers c WHERE c.reputation > $1 ORDER BY c.reputation DESC LIMIT 5
```

As you can see, the `@Limit(10)` annotation will not override the `LIMIT` defined explicitly in the raw query. As well, `@Hint("CustomerIdx")` annotation does not override the `HINT` explicitly defined in the raw query. Finally, the `@Trace` annotation is redundant and has no additional effect.

NOTE

The "ReputationIdx" Index is probably not the most sensible index given the number of Customers who will possibly have the same value for their reputation, which will effectively reduce the effectiveness of the index. Please choose indexes and other optimizations wisely as an improper or poorly choosen index and have the opposite effect on your performance given the overhead in maintaining the index. The "ReputationIdx" was only used to serve the purpose of the example.

Chapter 10. Annotation Support for Function Execution

10.1. Introduction

Spring Data GemFire 1.3.0 introduces annotation support to simplify working with GemFire Function Execution. The GemFire API provides classes to implement and register Functions deployed to Cache servers that may be invoked remotely by member applications, typically cache clients. Functions may execute in parallel, distributed among multiple servers, combining results in a map-reduce pattern, or may be targeted at a single server. A Function execution may be also be targeted to a specific Region.

GemFire also provides APIs to support remote execution of Functions targeted to various defined scopes (Region, member groups, servers, etc.) and the ability to aggregate results. The API also provides certain runtime options. The implementation and execution of remote Functions, as with any RPC protocol, requires some boilerplate code. Spring Data GemFire, true to Spring's core value proposition, aims to hide the mechanics of remote Function execution and allow developers to focus on POJO programming and business logic. To this end, Spring Data GemFire introduces annotations to declaratively register public methods as GemFire Functions, and the ability to invoke registered Functions remotely via annotated interfaces.

10.2. Implementation vs Execution

There are two separate concerns to address. First is the Function implementation (server) which must interact with the FunctionContext to obtain the invocation arguments, the ResultsSender and other execution context information. The Function implementation typically accesses the Cache and or Region and is typically registered with the FunctionService under a unique Id. The application invoking a Function (the client) does not depend on the implementation. To invoke a Function remotely, the application instantiates an Execution providing the Function ID, invocation arguments, the Function target or scope (Region, server, servers, member, members). If the Function produces a result, the invoker uses a ResultCollector to aggregate and acquire the execution results. In certain scenarios, a custom ResultCollector implementation is required and may be registered with the Execution.

NOTE	'Client' and 'Server' are used here in the context of Function execution which may have a different meaning than client and server in a client-server Cache topology. While it is common for a member with a Client Cache to invoke a Function on one or more Cache Server members it is also possible to execute Functions in a peer-to-peer (P2P) configuration

10.3. Implementing a Function

Using GemFire APIs, the FunctionContext provides a runtime invocation context including the client's calling arguments and a ResultSender interface to send results back to the client. Additionally, if the Function is executed on a Region, the FunctionContext is an instance of

RegionFunctionContext which provides additional context such as the target Region and any Filter (set of specific keys) associated with the Execution. If the Region is a PARTITION Region, the Function should use the PartitionRegionHelper to extract only the local data.

Using Spring, a developer can write a simple POJO and enable the Spring container to bind one or more of it's public methods to a Function. The signature for a POJO method intended to be used as a Function must generally conform to the the client's execution arguments. However, in the case of a Region execution, the Region data must also be provided (presumably the data held in the local partition if the Region is a PARTITION Region). Additionally the Function may require the Filter that was applied, if any. This suggests that the client and server may share a contract for the calling arguments but that the method signature may include additional parameters to pass values provided by the FunctionContext. One possibility is that the client and server share a common interface, but this is not required. The only constraint is that the method signature includes the same sequence of calling arguments with which the Function was invoked after the additional parameters are resolved.

For example, suppose the client provides a String and int as the calling arguments. These are provided by the FunctionContext as an array:

```
Object[] args = new Object[]{"hello", 123}
```

Then the Spring container should be able to bind to any method signature similar to the following. Let's ignore the return type for the moment:

```
public Object method1(String s1, int i2) {...}
public Object method2(Map<?,?> data, String s1, int i2) {...}
public Object method3(String s1, Map<?,?>data, int i2) {...}
public Object method4(String s1, Map<?,?> data, Set<?> filter, int i2) {...}
public void method4(String s1, Set<?> filter, int i2, Region<?,?> data) {...}
public void method5(String s1, ResultSender rs, int i2);
public void method6(FunctionContest fc);
```

The general rule is that once any additional arguments, i.e. Region data and Filter, are resolved, the remaining arguments must correspond exactly, in order and type, to the expected calling parameters. The method's return type must be void or a type that may be serialized (either java.io.Serializable, DataSerializable, or PDX serializable). The latter is also a requirement for the calling arguments. The Region data should normally be defined as a Map, to facilitate unit testing, but may also be of type Region if necessary. As shown in the example above, it is also valid to pass the FunctionContext itself, or the ResultSender, if you need to control how the results are returned to the client.

10.3.1. Annotations for Function Implementation

The following example illustrates how annotations are used to expose a POJO as a GemFire Function:

```
@Component
public class ApplicationFunctions {

   @GemfireFunction
   public String function1(String value, @RegionData Map<?,?> data, int i2) { ... }

   @GemfireFunction("myFunction", HA=true, optimizedForWrite=true, batchSize=100)
   public List<String> function2(String value, @RegionData Map<?,?> data, int i2,
@Filter Set<?> keys) { ... }

   @GemfireFunction(hasResult=true)
   public void functionWithContext(FunctionContext functionContext) { ... }

}
```

Note that the class itself must be registered as a Spring bean. Here the `@Component` annotation is used, but you may register the bean by any method provided by Spring (e.g. XML configuration or Java configuration class). This allows the Spring container to create an instance of this class and wrap it in a PojoFunctionWrapper (PFW). Spring creates one PFW instance for each method annotated with `@GemfireFunction`. Each will all share the same target object instance to invoke the corresponding method.

NOTE	The fact that the Function class is a Spring bean may offer other benefits since it shares the ApplicationContext with GemFire components such as a Cache and Regions. These may be injected into the class if necessary.

Spring creates the wrapper class and registers the Function with GemFire's Function Service. The Function id used to register the Functions must be unique. By convention it defaults to the simple (unqualified) method name. Note that this annotation also provides configuration attributes, `HA` and `optimizedForWrite` which correspond to properties defined by GemFire's Function interface. If the method's return type is void, then the `hasResult` property is automatically set to `false`; otherwise it is set to `true`.

For `void` return types, the annotation provides a `hasResult` attribute that can be set to true to override this convention, as shown in the `functionWithContext` method above. Presumably, the intention is to use the ResultSender directly to send results to the caller.

The PFW implements GemFire's Function interface, binds the method parameters, and invokes the target method in its `execute()` method. It also sends the method's return value using the ResultSender.

Batching Results

If the return type is a Collection or Array, then some consideration must be given to how the results are returned. By default, the PFW returns the entire Collection at once. If the number of items is large, this may incur a performance penalty. To divide the payload into small sections (sometimes called chunking), you can set the `batchSize` attribute, as illustrated in `function2`, above.

NOTE	If you need more control of the ResultSender, especially if the method itself would use too much memory to create the Collection, you can pass the ResultSender, or access it via the FunctionContext, to use it directly within the method.

Enabling Annotation Processing

In accordance with Spring standards, you must explicitly activate annotation processing for @GemfireFunction using XML:

```
<gfe:annotation-driven/>
```

or by annotating a Java configuration class:

```
@EnableGemfireFunctions
```

10.4. Executing a Function

A process invoking a remote Function needs to provide calling arguments, a Function id, the execution target (onRegion, onServers, onServer, onMember, onMembers) and optionally a Filter set. All a developer need do is define an interface supported by annotations. Spring will create a dynamic proxy for the interface which will use the FunctionService to create an Execution, invoke the Execution and coerce the results to a defined return type, if necessary. This technique is very similar to the way Spring Data Repositories work, thus some of the configuration and concepts should be familiar. Generally a single interface definition maps to multiple Function executions, one corresponding to each method defined in the interface.

10.4.1. Annotations for Function Execution

To support client-side Function execution, the following annotations are provided: `@OnRegion`, `@OnServer`, `@OnServers`, `@OnMember`, `@OnMembers`. These correspond to the Execution implementations GemFire's FunctionService provides. Each annotation exposes the appropriate attributes. These annotations also provide an optional `resultCollector` attribute whose value is the name of a Spring bean implementing ResultCollector to use for the execution.

NOTE	The proxy interface binds all declared methods to the same execution configuration. Although it is expected that single method interfaces will be common, all methods in the interface are backed by the same proxy instance and therefore all share the same configuration.

Here are some examples:

```
@OnRegion(region="someRegion", resultCollector="myCollector")
public interface FunctionExecution {

     @FunctionId("function1")
     String doIt(String s1, int i2);

     String getString(Object arg1, @Filter Set<Object> keys) ;

}
```

By default, the Function id is the simple (unqualified) method name. `@FunctionId` is used to bind this invocation to a different Function id.

Enabling Annotation Processing

The client-side uses Spring's component scanning capability to discover annotated interfaces. To enable Function execution annotation processing, you can use XML:

```
<gfe-data:function-executions base-package="org.example.myapp.functions"/>
```

Note that the `function-executions` element is provided in the `gfe-data` namespace. The `base-package` attribute is required to avoid scanning the entire classpath. Additional filters are provided as described in the Spring reference.

Optionally, a developer can annotate her Java configuration class:

```
@EnableGemfireFunctionExecutions(basePackages = "org.example.myapp.functions")
```

10.5. Programmatic Function Execution

Using the annotated interface as described in the previous section, simply wire your interface into a bean that will invoke the Function:

```
@Component
 public class MyApp {

    @Autowired FunctionExecution functionExecution;

    public void doSomething() {
         functionExecution.doIt("hello", 123);
    }

}
```

Alternately, you can use a Function Execution template directly. For example

`GemfireOnRegionFunctionTemplate` creates an `onRegion` Function execution. For example:

```
Set<?,?> myFilter = getFilter();
Region<?,?> myRegion = getRegion();
GemfireOnRegionOperations template = new GemfireOnRegionFunctionTemplate(myRegion);
String result = template.executeAndExtract("someFunction",myFilter,"hello","world"
,1234);
```

Internally, Function executions always return a List. `executeAndExtract` assumes a singleton List containing the result and will attempt to coerce that value into the requested type. There is also an `execute` method that returns the List itself. The first parameter is the Function id. The Filter argument is optional. The following arguments are a variable argument List.

10.6. Function Execution with PDX

When using Spring Data GemFire's Function annotation support combined with GemFire's PDX serialization, there are a few logistical things to keep in mind.

As explained above, and by way of example, typically developers will define GemFire Functions using POJO classes annotated with Spring Data GemFire Function annotations as so...

```
public class OrderFunctions {

  @GemfireFunction(...)
  Order process(@RegionData data, Order order, OrderSource orderSourceEnum, Integer
count);

}
```

NOTE the Integer count parameter is an arbitrary argument as is the separation of the Order and OrderSource Enum, which might be logical to combine. However, the arguments were setup this way to demonstrate the problem with Function executions in the context of PDX.

Your Order and OrderSource enum might be as follows...

```
public class Order ... {

  private Long orderNumber;
  private Calendar orderDateTime;
  private Customer customer;
  private List<Item> items

  ...
}


public enum OrderSource {
  ONLINE,
  PHONE,
  POINT_OF_SALE
  ...
}
```

Of course, a developer may define a Function Execution interface to call the 'process' GemFire Server Function...

```
@OnServer
public interface OrderProcessingFunctions {
  Order process(Order order, OrderSource orderSourceEnum, Integer count);
}
```

Clearly, this `process(..)` Order Function is being called from a client-side, client Cache (`<gfe:client-cache/>`) member-based application. This means that the Function arguments must be serializable. The same is true when invoking peer-to-peer member Functions (`@OnMember(s)) between peers in the cluster. Any form of `distribution` requires the data transmitted between client and server, or peers to be serializable.

Now, if the developer has configured GemFire to use PDX for serialization (instead of Java serialization, for instance) it is common for developers to set the `read-serialized` attribute to **true** on the GemFire server(s)...

```
<gfe:cache ⋯ pdx-read-serialized="true"/>
```

This causes all values read from the Cache (i.e. Regions) as well as information passed between client and servers, or peers to remain in serialized form, include, but not limited to Function arguments.

GemFire will only serialize application domain object types that you have specifically configured (registered), either using GemFire's ReflectionBasedAutoSerializer, or specifically (and recommended) using a "custom" GemFire PdxSerializer for your application domain types.

What is less than apparent, is that GemFire automatically handles Java Enum types regardless of whether they are explicitly configured (registered with a `ReflectionBasedAutoSerializer` regex

pattern to the `classes` parameter, or handled by a "custom" GemFire `PdxSerializer`) or not, and despite the fact that Java Enums implement `java.io.Serializable`.

So, when a developer has `pdx-read-serialized` set to **true** on the GemFire Servers on which the GemFire Functions (including Spring Data GemFire registered, Function annotated POJO classes), then the developer may encounter surprising behavior when invoking the Function Execution.

What the developer may pass as arguments when invoking the Function is...

```
  orderProcessingFunctions.process(new Order(123, customer, Calendar.getInstance(),
items), OrderSource.ONLINE, 400);
```

But, in actuality, what GemFire executes the Function on the Server is...

```
  process(regionData, order:PdxInstance, :PdxInstanceEnum, 400);
```

Notice that the `Order` and `OrderSource` have passed to the Function as PDX instances. Again, this is all because `read-serialized` is set to true on the GemFire Server, which may be necessary in cases where the GemFire Servers are interacting with multiple different client types (e.g. native clients).

This flies in the face of Spring Data GemFire's, "strongly-typed", Function annotated POJO class method signatures, as the developer is expecting application domain object types (not PDX serialized objects).

So, as of Spring Data GemFire (SDG) **1.6**, SDG introduces enhanced Function support to automatically convert method arguments that are of type PDX to the desired application domain object types when the developer of the Function expects his Function arguments to be "strongly-typed".

However, this also requires the developer to explicitly register a GemFire `PdxSerializer` on the GemFire Servers where the SDG annotated POJO Function is registered and used, e.g. ...

```
<bean id="customPdxSerializer" class="x.y.z.serialization.pdx.MyCustomPdxSerializer"/>

<gfe:cache ... pdx-serializer-ref="customPdxSerializeer" pdx-read-serialized="true"/>
```

Alternatively, a developer my use GemFire's ReflectionBasedAutoSerializer. Of course, it is recommend to use a "custom" `PdxSerializer` where possible to maintain finer grained control over your serialization strategy.

Finally, Spring Data GemFire is careful not to convert your Function arguments if you really want to treat your Function arguments generically, or as one of GemFire's PDX types...

```
@GemfireFunction
public Object genericFunction(String value, Object domainObject, PdxInstanceEnum enum)
{
 ...
}
```

Spring Data GemFire will only convert PDX type data to corresponding application domain object types if and only if the corresponding application domain object types are on the classpath the the Function annotated POJO method expects it.

For a good example of "custom", "composed" application-specific GemFire `PdxSerializers` as well as appropriate POJO Function parameter type handling based on the method signature, see Spring Data GemFire's ClientCacheFunctionExecutionWithPdxIntegrationTest class.

Chapter 11. Bootstrapping a Spring ApplicationContext in GemFire

11.1. Introduction

Normally, a Spring-based application will bootstrap GemFire using Spring Data GemFire's XML namespace. Just by specifying a `<gfe:cache/>` element in Spring Data GemFire configuration meta-data, a single, peer GemFire Cache instance will be created and initialized with default settings in the same JVM process as your application.

However, sometimes it is a requirement, perhaps imposed by your IT operations team, that GemFire must be fully managed and operated using the provided GemFire tool suite, such as with Gfsh. Using **Gfsh**, even though the application and GemFire will share the same JVM process, GemFire will bootstrap your Spring application context rather than the other way around. So, using this approach GemFire, instead of an application server, or a Java main class using Spring Boot, will bootstrap and host your application.

Keep in mind, however, that GemFire is not an application server. In addition, there are limitations to using this approach where GemFire Cache configuration is concerned.

11.2. Using GemFire to Bootstrap a Spring Context Started with Gfsh

In order to bootstrap a Spring application context in GemFire when starting a GemFire Server process using Gfsh, a user must make use of GemFire's Initalizer functionality. An **Initializer** can be used to specify a callback application that is launched after the Cache is initialized by GemFire.

An **Initializer** is specified within an initializer element using a minimal snippet of GemFire's native configuration meta-data inside a `cache.xml` file. The `cache.xml` file is required in order to bootstrap the Spring application context, much like a minimal snippet of Spring XML config is needed to bootstrap a Spring application context configured with component scanning (e.g. `<context:component-scan base-packages="⋯"/>`)

As of Spring Data GemFire 1.4, such an **Initializer** is already conveniently provided by the framework, the `org.springframework.data.gemfire.support.SpringContextBootstrappingInitializer`. The typical, yet minimal configuration for this class inside GemFire's `cache.xml` file will look like the following:

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC  "-//GemStone Systems, Inc.//GemFire Declarative Caching
7.0//EN"
  "http://www.gemstone.com/dtd/cache7_0.dtd">

<cache>
  <initializer>
    <class-
name>org.springframework.data.gemfire.support.SpringContextBootstrappingInitializer</c
lass-name>
    <parameter name="contextConfigLocations">
      <string>classpath:application-context.xml</string>
    </parameter>
  </initializer>
</cache>
```

The `SpringContextBootstrappingInitializer` class follows similar conventions as Spring's ContextLoaderListener class for bootstrapping a Spring context inside a Web Application, where application context configuration files are specified with the `contextConfigLocations` Servlet Context Parameter. In addition, the `SpringContextBootstrappingInitializer` class can also be used with a `basePackages` parameter to specify a comma-separated list of base package containing the appropriately annotated application components that the Spring container will search using component scanning and create Spring beans for:

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC  "-//GemStone Systems, Inc.//GemFire Declarative Caching
7.0//EN"
  "http://www.gemstone.com/dtd/cache7_0.dtd">

<cache>
  <initializer>
    <class-
name>org.springframework.data.gemfire.support.SpringContextBootstrappingInitializer</c
lass-name>
    <parameter name="basePackages">
      <string>org.mycompany.myapp.services,org.mycompany.myapp.dao,...</string>
    </parameter>
  </initializer>
</cache>
```

Then, with a properly configured and constructed `CLASSPATH` along with the `cache.xml` file shown above specified as a command-line option when starting a GemFire Server in Gfsh, the command-line would be:

```
gfsh>start server --name=Server1 --log-level=config ...
            --classpath="/path/to/spring-data-gemfire
-1.4.0.jar:/path/to/application/classes.jar"
            --cache-xml-file="/path/to/gemfire/cache.xml"
```

The `application-context.xml` can be any valid Spring context configuration meta-data including all the SDG namespace elements. The only limitation with this approach is that the GemFire Cache cannot be configured using the Spring Data GemFire namespace. In other words, none of the `<gfe:cache/>` element attributes, such as `cache-xml-location`, `properties-ref`, `critical-heap-percentage`, `pdx-serializer-ref`, `lock-lease`, etc can be specified. If used, these attributes will be ignored. The main reason for this is that GemFire itself has already created an initialized the Cache before the **Initializer** gets invoked. As such, the Cache will already exist and since it is a "Singleton", it cannot be re-initialized or have any of it's configuration augmented.

11.3. Lazy-Wiring GemFire Components

Spring Data GemFire already provides existing support for wiring GemFire components (such as CacheListeners, CacheLoaders or CacheWriters) that are declared and created by GemFire in `cache.xml` using the `WiringDeclarableSupport` class as described in Configuration using auto-wiring and annotations. However, this only works when Spring does the bootstrapping (i.e. bootstraps GemFire). When your Spring application context is the one bootstrapped by GemFire, then these GemFire components go unnoticed since the Spring application context does not even exist yet! The Spring application context will not get created until GemFire calls the **Initializer**, which occurs after all the other GemFire components and configuration have already been created and initialized.

So, in order to solve this problem, a new `LazyWiringDeclarableSupport` class was introduced, that is, in a sense, Spring application context aware. The intention of this abstract base class is that any implementing class will register itself to be configured by the Spring application context created by GemFire after the **Initializer** is called. In essence, this give your GemFire managed component a chance to be configured and auto-wired with Spring beans defined in the Spring application context.

In order for your GemFire application component to be auto-wired by the Spring container, create a application class that extends the `LazyWiringDeclarableSupport` and annotate any class member that needs to be provided as a Spring bean dependency, similar to:

```
public static final class UserDataSourceCacheLoader extends
LazyWiringDeclarableSupport implements CacheLoader<String, User> {

  @Autowired
  private DataSource userDataSource;

  ...
}
```

As implied by the CacheLoader example above, you might necessarily (although, rare) have defined both a Region and CacheListener component in GemFire `cache.xml`. The CacheLoader may need access to an application DAO, or perhaps Spring application context defined JDBC Data Source for loading "Users" into a GemFire Cache `REPLICATE` Region on start. Of course, one should be careful in mixing the different life-cycles of GemFire and the Spring Container together in this manner as not all use cases and scenarios are supported. The GemFire `cache.xml` configuration would be similar to the following (which comes from SDG's test suite):

```
<?xml version="1.0"?>
<!DOCTYPE cache PUBLIC  "-//GemStone Systems, Inc.//GemFire Declarative Caching
7.0//EN"
  "http://www.gemstone.com/dtd/cache7_0.dtd">

<cache>
  <region name="Users" refid="REPLICATE">
    <region-attributes initial-capacity="101" load-factor="0.85">
      <key-constraint>java.lang.String</key-constraint>
      <value-constraint>
org.springframework.data.gemfire.repository.sample.User</value-constraint>
      <cache-loader>
        <class-
name>org.springframework.data.gemfire.support.SpringContextBootstrappingInitializerInt
egrationTest$UserDataStoreCacheLoader</class-name>
      </cache-loader>
    </region-attributes>
  </region>
  <initializer>
    <class-
name>org.springframework.data.gemfire.support.SpringContextBootstrappingInitializer</c
lass-name>
    <parameter name="basePackages">
      <string>org.springframework.data.gemfire.support.sample</string>
    </parameter>
  </initializer>
</cache>
```

Chapter 12. Sample Applications

NOTE Sample applications are now maintained in the Spring Data GemFire Examples repository.

The Spring Data GemFire project also includes one sample application. Named "Hello World", the sample demonstrates how to configure and use GemFire inside a Spring application. At runtime, the sample offers a **shell** to the user allowing him to run various commands against the grid. It provides an excellent starting point for users unfamiliar with the essential components or the Spring and GemFire concepts.

The sample is bundled with the distribution and is Maven-based. One can easily import them into any Maven-aware IDE (such as Spring Tool Suite) or run them from the command-line.

12.1. Hello World

The Hello World sample demonstrates the core functionality of the Spring GemFire project. It bootstraps GemFire, configures it, executes arbitrary commands against it and shuts it down when the application exits. Multiple instances can be started at the same time as they will work with each other sharing data without any user intervention.

NOTE

Running under Linux

If you experience networking problems when starting GemFire or the samples, try adding the following system property `java.net.preferIPv4Stack=true` to the command line (insert `-Djava.net.preferIPv4Stack=true`). For an alternative (global) fix especially on Ubuntu see this link

12.1.1. Starting and stopping the sample

Hello World is designed as a stand-alone java application. It features a `Main` class which can be started either from your IDE of choice (in Eclipse/STS through `Run As/Java Application`) or from the command line through Maven using `mvn exec:java`. One can also use `java` directly on the resulting artifact if the classpath is properly set.

To stop the sample, simply type `exit` at the command line or press `Ctrl+C` to stop the VM and shutdown the Spring container.

12.1.2. Using the sample

Once started, the sample will create a shared data grid and allow the user to issue commands against it. The output will likely look as follows:

```
INFO: Created GemFire Cache [Spring GemFire World] v. X.Y.Z
INFO: Created new cache region [myWorld]
INFO: Member xxxxxx:50694/51611 connecting to region [myWorld]
Hello World!
Want to interact with the world ? ...
Supported commands are:

get <key> - retrieves an entry (by key) from the grid
put <key> <value> - puts a new entry into the grid
remove <key> - removes an entry (by key) from the grid
...
```

For example to add new items to the grid one can use:

```
-> Bold Section qName:emphasis level:5, chunks:[put 1 unu] attrs:[role:bold]
INFO: Added [1=unu] to the cache
null
-> Bold Section qName:emphasis level:5, chunks:[put 1 one] attrs:[role:bold]
INFO: Updated [1] from [unu] to [one]
unu
-> Bold Section qName:emphasis level:5, chunks:[size] attrs:[role:bold]
1
-> Bold Section qName:emphasis level:5, chunks:[put 2 two] attrs:[role:bold]
INFO: Added [2=two] to the cache
null
-> Bold Section qName:emphasis level:5, chunks:[size] attrs:[role:bold]
2
```

Multiple instances can be created at the same time. Once started, the new VMs automatically see the existing region and its information:

```
INFO: Connected to Distributed System ['Spring GemFire World'=xxxx:56218/49320@yyyyy]
Hello World!
...

-> Bold Section qName:emphasis level:5, chunks:[size] attrs:[role:bold]
2
-> Bold Section qName:emphasis level:5, chunks:[map] attrs:[role:bold]
[2=two] [1=one]
-> Bold Section qName:emphasis level:5, chunks:[query length = 3] attrs:[role:bold]
[one, two]
```

Experiment with the example, start (and stop) as many instances as you want, run various commands in one instance and see how the others react. To preserve data, at least one instance needs to be alive all times - if all instances are shutdown, the grid data is completely destroyed (in this example - to preserve data between runs, see the GemFire documentations).

12.1.3. Hello World Sample Explained

Hello World uses both Spring XML and annotations for its configuration. The initial boostrapping configuration is `app-context.xml` which includes the cache configuration, defined under `cache-context.xml` file and performs classpath scanning for Spring components. The cache configuration defines the GemFire cache, region and for illustrative purposes a simple cache listener that acts as a logger.

The main **beans** are `HelloWorld` and `CommandProcessor` which rely on the `GemfireTemplate` to interact with the distributed fabric. Both classes use annotations to define their dependency and life-cycle callbacks.

Other Resources

In addition to this reference documentation, there are a number of other resources that may help you learn how to use GemFire and Spring framework. These additional, third-party resources are enumerated in this section.

Chapter 13. Useful Links

- Spring Data GemFire Home Page
- Pivotal GemFire Home Page
- Pivotal GemFire Documentation
- Pivotal GemFire Knowledge Base
- Pivotal GemFire Community Home Page
- VMWare vFabric GemFire Community Home Page
- Spring Data GemFire Forum (StackOverflow)
- Spring Data GemFire Forum (spring.io archive)