Spring Data for Apache Geode - Reference Guide

Costin Leau, David Turanski, John Blum, Oliver Gierke

Version 2.0.5.RELEASE, 2018-02-28

Table of Contents

Preface . . . 2
1. Introduction . . . 3
2. Requirements . . . 4
3. New Features . . . 5
3.1. New in the 1.0.0.RELEASE . . . 5
3.2. New in the 1.1.0.RELEASE . . . 5
3.3. New in the 2.0.0.RELEASE . . . 5
Reference Guide . . . 7
4. Document Structure . . . 8
5. Bootstrapping Apache Geode with the Spring container . . . 9
5.1. Advantages of using Spring over Apache Geode `cache.xml` . . . 9
5.2. Using the Core Namespace . . . 9
5.3. Using the Data Access Namespace . . . 11
5.3.1. An Easy Way to Connect to Geode . . . 11
5.4. Configuring a Cache . . . 12
5.4.1. Advanced Cache Configuration . . . 14
5.4.2. Configuring a Geode CacheServer . . . 17
5.4.3. Configuring a Geode ClientCache . . . 18
5.5. Configuring a Region . . . 21
5.5.1. Using an externally configured Region . . . 21
5.5.2. Auto Region Lookup . . . 22
5.5.3. Configuring Regions . . . 23
5.5.4. Compression . . . 27
5.5.5. Subregions . . . 27
5.5.6. Region Templates . . . 28
5.5.7. Data Eviction (with Overflow) . . . 33
5.5.8. Data Expiration . . . 33
5.5.9. Data Persistence . . . 38
5.5.10. Subscription Policy . . . 38
5.5.11. Local Region . . . 38
5.5.12. Replicated Region . . . 39
5.5.13. Partitioned Region . . . 39
5.5.14. Client Region . . . 41
5.5.15. JSON Support . . . 44
5.6. Configuring an Index . . . 44
5.6.1. Defining Indexes . . . 46
5.6.2. `IgnoreIfExists` and `Override` . . . 46
5.7. Configuring a DiskStore . . . 49

5.8. Configuring the Snapshot Service . . . 50
5.8.1. Snapshot Location . . . 51
5.8.2. Snapshot Filters . . . 52
5.8.3. Snapshot Events . . . 53
5.9. Configuring the Function Service . . . 54
5.10. Configuring WAN Gateways . . . 55
5.10.1. WAN Configuration in GemFire 7.0 . . . 55
6. Bootstrapping Apache Geode using Spring Annotations . . . 58
6.1. Introduction . . . 58
6.2. Bootstrapping Apache Geode applications with Spring . . . 59
6.3. Going in-detail on *client/server* applications . . . 60
6.4. Runtime configuration using `Configurers` . . . 63
6.5. Runtime configuration using `Properties` . . . 65
6.5.1. `Properties` of `Properties` . . . 67
6.6. Configuring embedded services . . . 67
6.6.1. Configuring an embedded Locator . . . 67
6.6.2. Configuring an embedded Manager . . . 69
6.6.3. Configuring the embedded HTTP Server . . . 70
6.6.4. Configuring the embedded Memcached Server (Gemcached) . . . 71
6.6.5. Configuring the embedded Redis Server . . . 71
6.7. Configuring Logging . . . 72
6.8. Configuring Statistics . . . 72
6.9. Configuring PDX . . . 73
6.10. Configuring SSL . . . 74
6.11. Configuring GemFire Properties . . . 75
6.12. Configuring Regions . . . 76
6.12.1. Configuring Type-specific Regions . . . 79
6.12.2. Configuring Eviction . . . 81
6.12.3. Configuring Expiration . . . 82
6.12.4. Configuring Compression . . . 83
6.12.5. Configuring Off-Heap . . . 84
6.12.6. Configuring Indexes . . . 84
6.12.7. Configuring Disk Stores . . . 88
6.13. Configuring Continuous Queries . . . 90
6.14. Configuring Spring's Cache Abstraction . . . 92
6.15. Configuring Cluster Configuration Push . . . 94
6.16. Configuring Security . . . 96
6.16.1. Configuring Server Security . . . 96
6.16.2. Configuring Client Security . . . 98
6.17. Configuration Tips . . . 99
6.18. Configuration Organization . . . 99

6.19. Additional Configuration-based Annotations . . . 100
6.20. Conclusion . . . 101
7. Working with Apache Geode APIs . . . 102
7.1. GemfireTemplate . . . 102
7.2. Exception Translation . . . 102
7.3. Local, Cache Transaction Management . . . 103
7.4. Global, JTA Transaction Management . . . 103
7.5. Continuous Query (CQ) . . . 107
7.5.1. Continuous Query Listener Container . . . 108
7.5.2. The `ContinuousQueryListener` and `ContinuousQueryListenerAdapter` . . . 108
7.6. Wiring `Declarable` Components . . . 111
7.6.1. Configuration using **template** bean definitions . . . 112
7.6.2. Configuration using auto-wiring and annotations . . . 114
7.7. Support for the Spring Cache Abstraction . . . 115
8. Working with Apache Geode Serialization . . . 119
8.1. Wiring deserialized instances . . . 119
8.2. Auto-generating custom `Instantiators` . . . 120
9. POJO mapping . . . 121
9.1. Entity Mapping . . . 121
9.1.1. Entity Mapping by Region Type . . . 122
9.1.2. Repository Mapping . . . 122
9.2. Mapping PDX Serializer . . . 123
10. Spring Data Geode Repositories . . . 125
10.1. Introduction . . . 125
10.2. Spring XML Configuration . . . 125
10.3. Spring Java-based Configuration . . . 125
10.4. Executing OQL Queries . . . 126
10.5. OQL Query Extensions using Annotations . . . 128
10.6. Query Post Processing . . . 130
11. Annotation Support for Function Execution . . . 135
11.1. Introduction . . . 135
11.2. Implementation vs Execution . . . 135
11.3. Implementing a Function . . . 136
11.3.1. Annotations for Function Implementation . . . 137
11.3.2. Batching Results . . . 138
11.3.3. Enabling Annotation Processing . . . 138
11.4. Executing a Function . . . 138
11.4.1. Annotations for Function Execution . . . 138
11.4.2. Enabling Annotation Processing . . . 139
11.5. Programmatic Function Execution . . . 139
11.6. Function Execution with PDX . . . 140

12. Apache Lucene Integration . . . 144
12.1. Lucene Template Data Accessors . . . 145
12.2. Annotation configuration support . . . 149
13. Bootstrapping a Spring ApplicationContext in Apache Geode . . . 150
13.1. Introduction . . . 150
13.2. Using Apache Geode to Bootstrap a Spring Context Started with Gfsh . . . 150
13.3. Lazy-Wiring GemFire Components . . . 152
14. Sample Applications . . . 155
14.1. Hello World . . . 155
14.1.1. Starting and stopping the sample . . . 155
14.1.2. Using the sample . . . 155
14.1.3. Hello World Sample Explained . . . 157
Resources . . . 158
15. Useful Links . . . 159
Appendices . . . 160
Appendix A: Namespace reference . . . 161
The <repositories /> element . . . 161
Appendix B: Populators namespace reference . . . 162
The <populator /> element . . . 162
Appendix C: Repository query keywords . . . 163
Supported query keywords . . . 163
Appendix D: Repository query return types . . . 164
Supported query return types . . . 164
Appendix E: Spring Data Geode Schema . . . 166

Preface

Spring Data Geode focuses on integrating the *Spring Framework's* powerful, non-invasive programming model and concepts with Apache Geode to simplify configuration and development of Java applications using Geode.

This document assumes the reader already has a basic familiarity with the *Spring Framework* and Apache Geode concepts and APIs.

While every effort has been made to ensure this documentation is comprehensive and complete, with no errors, some topics are beyond the scope of this document and may require more explanation (e.g. data distribution management with partitioning for HA while still preserving consistency). Additionally, some typos might have crept in. If you do spot mistakes or even more serious errors and you can spare a few cycles, please do bring these issues to the attention of the *Spring Data Geode* team by raising an appropriate issue.

Thank you.

Chapter 1. Introduction

This reference guide for *Spring Data Geode* explains how to use the *Spring Framework* to configure and develop applications with Apache Geode. It presents the basic concepts, semantics and provides numerous examples to help you get started.

Chapter 2. Requirements

Spring Data Geode requires JDK 8.0, Spring Framework 5 and Apache Geode 1.2.0.

Chapter 3. New Features

NOTE	As of version `2.0.0`, *Spring Data Geode* is now a top-level module in the Spring Data project.

3.1. New in the 1.0.0.RELEASE

- Upgrades to Apache Geode 1.0.0-incubating (GA) release.
- Upgrades to Spring Framework 4.3.4.RELEASE.
- Significant additions to the new Annotation-based configuration model.
- Support for CDI.
- Ability to configure Apache Geode's Off-Heap memory support.
- Fix for premature destruction of client Pools before the Region's configured to use these Pools.
- Support Repositories with multiple SD modules on the classpath.
- Support for `forwardExpirationDestroy` in the `AsyncEventQueueFactoryBean` API and XML namespace.
- Handle case-insensitive OQL queries defined as Repository query methods.
- Enable explicit Cache names referring to Regions to be specified when using GemfireCacheManager.
- Fix for ordered GemfireRepository.findAll(Sort) queries.
- GemfireCache.evict(key) now calls Region.remove(key).
- Fix RegionNotFoundException when executing Repository queries on client Regions configured with a Pool targeted for a specific server group.
- Geode package namespace changed from com.gemstone.gemfire to org.apache.geode.
- Support for the Geode Integrated Security framework.

3.2. New in the 1.1.0.RELEASE

- Upgrades to Aapche Geode 1.1.0 (GA) release.
- Upgrades to Spring Framework 4.3.7.RELEASE.
- Upgrades to Spring Data Commons Ingalls-SR1.
- Additional improvements in the new Annotation-based configuration model.
- Support Apache Geode's Apache Lucene Integration.

3.3. New in the 2.0.0.RELEASE

- Upgrades to Apache Geode 1.2.0 (GA) release.
- Upgrades to Spring Framework 5.0.0.RELEASE.

- Upgrades to Spring Data Commons Kay.
- Spring Data Geode joins the Spring Data Release Train (Kay) as a new Spring Data module.
- Additional improvements in the new Annotation-based configuration model.
- Support Apache Geode's Apache Lucene Integration.

Reference Guide

Chapter 4. Document Structure

The following chapters explain the core functionality offered by *Spring Data Geode* for Apache Geode.

Bootstrapping Apache Geode with the Spring container describes the configuration support provided for bootstrapping, configuring, initializing and accessing Apache Geode Caches, Regions, and related Distributed System components.

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

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

POJO mapping describes persistence mapping for POJOs stored in Apache Geode using *Spring Data*.

Spring Data Geode Repositories describes how to create and use *Spring Data Repositories* to access data in Apache Geode.

Annotation Support for Function Execution describes how to create and use Apache Geode Functions using Annotations.

Bootstrapping a Spring ApplicationContext in Apache Geode describes how to bootstrap a *Spring* `ApplicationContext` running in an Apache Geode server using *Gfsh*.

Sample Applications describes the examples provided with the distribution to illustrate the various features available in *Spring Data Geode*.

Chapter 5. Bootstrapping Apache Geode with the Spring container

Spring Data Geode provides full configuration and initialization of the Apache Geode In-Memory Data Grid (IMDG) using the *Spring* IoC container. The framework includes several classes to help simplify the configuration of Apache Geode components including: Caches, Regions, Indexes, DiskStores, Functions, WAN Gateways, persistence backup along with several other Distributed System components in order to support a variety of use cases with minimal effort.

NOTE	This section assumes basic familiarity with Apache Geode. For more information, see the Apache Geode product documentation.

5.1. Advantages of using Spring over Apache Geode `cache.xml`

Spring Data Geode's XML namespace supports full configuration of the Apache Geode In-Memory Data Grid (IMDG). The XML namespace is the preferred way to configure Apache Geode in a *Spring* context in order to properly manage Geode's lifecycle inside the *Spring* container. While support for Geode's native `cache.xml` persists for legacy reasons, Geode application developers are encouraged to do everything in *Spring* XML to 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 Geode* makes extensive use of *Spring's* `FactoryBean` pattern to simplify the creation, configuration and initialization of Geode components.

Apache Geode provides several callback interfaces, such as `CacheListener`, `CacheLoader` and `CacheWriter`, 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 Geode components. This is a significant improvement over native `cache.xml`, which provides relatively limited configuration options and requires callbacks to implement Geode'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 including code completion, pop-up annotations, and real time validation, making them easy to use.

5.2. Using the Core Namespace

To simplify configuration, *Spring Data Geode* provides a dedicated XML namespace for configuring core Apache Geode components. It is possible to configure beans directly using *Spring's* standard `<bean>` definition. However, 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 the appendix in the *Spring Framework* reference documentation.

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

To use the *Spring Data Geode* 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/geode"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/geode
http://www.springframework.org/schema/gemfire/spring-geode.xsd"> ③

  <bean id ... >

  <gfe:cache ...> ④

</beans>
```

① *Spring Data Geode* XML namespace prefix. Any name will do but through out this reference documentation, `gfe` will be used.

② The XML namespace prefix is mapped to the URI.

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

④ Example declaration using the XML namespace with the `gfe` prefix.

NOTE

It is possible to change the default namespace from `beans` to `gfe`. This is useful for XML configuration composed mainly of Geode 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/geode" ①
  xmlns:beans="http://www.springframework.org/schema/beans" 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/geode
http://www.springframework.org/schema/gemfire/spring-geode.xsd">

  <beans:bean id ... > ③

  <cache ...> ④

</beans>
```

① The default namespace declaration for this XML document points to the *Spring Data Geode* XML namespace.

② The `beans` namespace prefix declaration for *Spring's* raw bean definitions.

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

④ Bean declaration using the `gfe` namespace. Notice the lack of prefix since `gfe` is the default namespace.

5.3. Using the Data Access Namespace

In addition to the core XML namespace (`gfe`), *Spring Data Geode* provides a `gfe-data` XML namespace primarily intended to simplify the development of Apache Geode client applications. This namespace currently contains support for Geode Repositories and function execution as well as includes a `<datasource>` tag that offers a convenient way to connect to the Apache Geode data grid.

5.3.1. An Easy Way to Connect to Geode

For many applications, a basic connection to a Geode data grid using default values is sufficient. *Spring Data Geode's* `<datasource>` tag provides a simple way to access data. The data source creates a `ClientCache` and connection `Pool`. In addition, it will query the cluster servers for all existing root Regions and create an (empty) client Region proxy for each one.

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

The `<datasource>` tag is syntactically similar to `<gfe:pool>`. It may be configured with one or more nested `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 client Region beans for each Region defined on cluster 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="remotehost" port="1234"/>
</gfe-data:datasource>

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

5.4. Configuring a Cache

To use Apache Geode, a developer needs to either create a new `Cache` or connect to an existing one. With the current version of Geode, 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 peer cache member, appropriate in peer-to-peer (P2P) topologies and cache servers. A cache member can also be used in standalone applications and integration tests. However, in most typical production systems, most application processes will act as cache clients, creating a `ClientCache` instance instead. This is described in the sections Configuring a Geode ClientCache and Client Region.

A peer 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 Geode `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 Geode* 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 XML namespace elements. Also, you can easily override the cache's bean name using the `id` attribute:

```
<gfe:cache id="myCache"/>
```

A Geode `Cache` can be fully configured using Spring, however, Geode's native XML configuration file, `cache.xml`, is also supported. For situations where the Geode cache needs to be configured natively, simply provide a reference to the Geode XML configuration file using the `cache-xml-location` attribute:

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

In this example, if a cache needs to be created, it will use a 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 Geode 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/geode"
  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/geode
http://www.springframework.org/schema/gemfire/spring-geode.xsd
    http://www.springframework.org/schema/util
http://www.springframework.org/schema/util/spring-util.xsd">

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

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

</beans>
```

Using a properties file 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.

5.4.1. 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
    cache-xml-location=".."
    properties-ref=".."
    close="false"
    copy-on-read="true"
    critical-heap-percentage="90"
    eviction-heap-percentage="70"
    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-bean-factory-locator="true" ③
    use-cluster-configuration="false" ④
>

  <gfe:transaction-listener ref="myTransactionListener"/> ⑤

  <gfe:transaction-writer> ⑥
    <bean class="org.example.app.geode.transaction.TransactionWriter"/>
  </gfe:transaction-writer>

  <gfe:gateway-conflict-resolver ref="myGatewayConflictResolver"/> ⑦

  <gfe:dynamic-region-factory/> ⑧

  <gfe:jndi-binding jndi-name="myDataSource" type="ManagedDataSource"/> ⑨

</gfe:cache>
```

① Various cache options are supported by attributes. For further information regarding anything shown in this example, please consult the Geode product documentation. The `close` attribute determines whether the cache should be closed when the Spring application context is closed. The default is `true`, however, for use cases in which multiple application contexts use the cache (common in web applications), set this value to `false`.

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

③ Setting the `use-bean-factory-locator` attribute to `true` (defaults to `false`) is only applicable when both Spring (XML) configuration meta-data and Geode `cache.xml` is used to configure the Geode cache node (whether client or peer). This option allows Geode components (e.g. `CacheLoader`) expressed in `cache.xml` to be auto-wired with beans (e.g. `DataSource`) defined in the Spring application context. This option is typically used in conjunction with `cache-xml-location`.

④ Setting the `use-cluster-configuration` attribute to `true` (default is `false`) enables a Geode member to retrieve the common, shared Cluster-based configuration from a Locator. See the Geode product documentation for more details.

⑤ Example of a `TransactionListener` callback declaration using a bean reference. The referenced bean must implement TransactionListener. A `TransactionListener` can be implemented to handle transaction related events (e.g. afterCommit, afterRollback).

⑥ Example of a `TransactionWriter` callback declaration using an inner bean declaration. The bean must implement TransactionWriter. The `TransactionWriter` is a callback that is allowed to veto a transaction.

⑦ Example of a `GatewayConflictResolver` callback declaration using a bean reference. The referenced bean must implement http://geode.apache.org/releases/latest/javadoc/org/apache/geode/cache/util/GatewayConflictResolver.html [GatewayConflictResolver]. A `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.

⑧ Enable Geode's DynamicRegionFactory, which provides a distributed Region creation service.

⑨ Declares a JNDI binding to enlist an external DataSource in a Geode transaction.

Enabling PDX Serialization

The example above includes a number of attributes related to Geode'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 `PdxSerializer` which is specified via the `pdx-serializer` attribute. Geode provides an implementing class `org.apache.geode.pdx.ReflectionBasedAutoSerializer` that uses Java Reflection, 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 `PdxSerializer` interface.

More information on serialization support can be found in Working with Apache Geode Serialization

Enabling auto-reconnect

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

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

The main reason for this is that most Geode applications use references to the Geode cache or

Regions in order to perform data access operations. These references are "injected" by the Spring container into application components (e.g. DAOs or Repositories) for use by the application. When a peer member is forcefully disconnected from the rest of the cluster, presumably because the peer member has become unresponsive or a network partition separates one or more peer members into a group too small to function as an independent distributed system, the peer member will shutdown and all Geode component references (e.g. Cache, Regions, etc) become invalid.

Essentially, the current forced-disconnect processing logic in each peer member dismantles the system from the ground up. The JGroups stack shuts down, the Distributed System is put in a shutdown state and finally, the Cache is closed. Effectively, all memory references become stale and are lost.

After being disconnected from the Distributed System a peer member enters a "reconnecting" state and periodically attempts to rejoin the Distributed System. If the peer member succeeds in reconnecting, the member rebuilds its "view" of the Distributed System from existing members and receives a new Distributed System ID. Additionally, all Cache, Regions and other Geode components are reconstructed. Therefore, all old references, which may have been injected into application by the Spring container are now stale and no longer valid.

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

Unfortunately, there is no way to be notified of a disconnect event, and subsequently, a reconnect event. If that were the case, the application developer would 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 Apache Geode is not recommended for peer cache Geode applications.

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

Using Cluster-based Configuration

Apache Geode's Cluster Configuration Service is a convenient way for any peer member joining the cluster to get a "consistent view" of the cluster by using the shared, persistent configuration maintained by a Locator. Using the Cluster-based Configuration ensures the peer member's configuration will be compatible with the Geode Distributed System when the member joins.

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

All Geode 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 Geode configuration meta-data and 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 Geode tools, like *Gfsh*, have their actions "recorded" when schema-like changes are made (e.g. `gfsh>create region --name=Example --type=PARTITION`), *Spring Data Geode's* configuration meta-data is not recorded. The same is true when using Geode's public Java API directly; it too is not recorded.

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

5.4.2. Configuring a Geode CacheServer

Spring Data Geode 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:context="http://www.springframework.org/schema/context"
  xmlns:gfe="http://www.springframework.org/schema/geode"
  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/context
http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/geode
http://www.springframework.org/schema/geode/spring-geode.xsd
">

  <gfe:cache/>

  <!-- Example depicting serveral Geode CacheServer configuration options -->
  <gfe:cache-server id="advanced-config" auto-startup="true"
       bind-address="localhost" host-name-for-clients="localhost" port=
"${geode.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 Geode* will start **after** the *Spring* 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.

5.4.3. Configuring a Geode ClientCache

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

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

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

`client-cache` supports many of the same options as the cache element. However, as opposed to a **full-fledged** peer cache member, a cache client 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 specific 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 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="myPool"/>

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

The `<client-cache>` element also has a `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.

Geode's DEFAULT Pool and Spring Data Geode Pool Definitions

If a Geode `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 Geode's ClientRegionShortcut (all `LOCAL_*` shortcuts).

However, if a 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 Geode* will resolve the references automatically for you.

For example:

```
<gfe:client-cache/>

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

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

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

Basically, *Spring Data Geode* resolves the above configuration to the following:

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

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

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

Geode still creates a Pool called "DEFAULT". *Spring Data Geode* will just cause the "DEFAULT" Pool to be initialized from the `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="${geode.locator.host}" port="${geode.locator.port}"/>
</gfe:pool>

<gfe:pool id="serverPool">
  <gfe:server host="${geode.server.host}" port="${geode.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 Geode client cache's "DEFAULT" Pool is initialized from "locatorPool" as specified with the `pool-name` attribute. There is no *Spring Data Geode*-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 Geode'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.

5.5. Configuring a Region

A Region is required to store and retrieve data from the cache. `org.apache.geode.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 application 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.

Geode implements the following types of Regions:

- **REPLICATE** - 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.
- **PARTITION** - Data is partitioned into buckets (sharded) among cache members that define the Region. This provides high read and write performance and is suitable for 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 REPLICATE or PARTITION Region hosted on cache servers in a cluster. 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 up-to-date (synchronized) with changes originating from remote processes that access the same server Region.

For more information about the various Region types and their capabilities as well as configuration options, please refer to Apache Geode's documentation on Region Types.

5.5.1. Using an externally configured Region

To reference Regions already configured in a Geode native `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 identified as `ordersRegion` for an existing Region named `Orders`, you can use the following bean definition:

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

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

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

CAUTION	If the Region does not exist, an initialization exception will be thrown. To configure new Regions, proceed to the appropriate sections below.

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

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

`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

"auto-lookup" allows all Regions defined in a Geode native `cache.xml` file to be imported into a *Spring* application context when using the`cache-xml-location` attribute on the `<gfe:cache>` element.

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

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

  <region name="Parent" refid="REPLICATE">
    <region name="Child" refid="REPLICATE"/>
  </region>

</cache>
```

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

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

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

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

Spring Data Geode will automatically create beans for all Geode Regions defined in `cache.xml` that

have not been explicitly added to the *Spring* context with explicit `<gfe:lookup-region>` bean declarations.

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

You may inject these "auto-looked-up" Regions like any other bean defined in the *Spring* application 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 example above is applicable when using *Spring's* `component-scan` functionality.

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

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

This ensures the Geode 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 Geode provides comprehensive support for configuring any type of Region via the following elements:

- LOCAL Region: `<local-region>`
- PARTITION Region: `<partitioned-region>`
- REPLICATE Region: `<replicated-region>`
- Client Region: `<client-region>`

Please see the Apache Geode documentation for a comprehensive description of Region Types.

Common Region Attributes

The following table lists attributes available for all Region types:

Table 1. Common Region Attributes

Name	Values	Description
cache-ref	Geode Cache bean reference	The name of the bean defining the Geode Cache (by default 'gemfireCache').
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.
close	boolean, default:false	Determines whether the Region should be closed at shutdown.
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 Geode's Data Policy	The Region's Data Policy. Note, not all Data Policies are supported for every Region type.
destroy	boolean, default:false	Determines 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	Determines whether Disk Store writes are synchronous.
id	Any valid bean name.	Will be the Region name by default if no `name` attribute is specified.
ignore-if-exists	boolean, default:false	Ignores this bean definition if the Region already exists in the cache, resulting in a lookup instead.
ignore-jta	boolean, default:false	Determines whether this Region will participate in JTA transactions.
index-update-type	synchronous or asynchronous, default:synchronous	Determines whether Indices will be updated synchronously or asynchronously on entry creation.
initial-capacity	integer, default:16	The initial memory allocation for the number of Region entries.
key-constraint	Any valid, fully-qualified Java class name.	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. If not specified, it will assume the value of the `id` attribute (a.k.a. bean name).
persistent	*boolean, default:false	Determines whether the Region will persist entries to local disk (Disk Store).
shortcut	See http://geode.apache.org/releases/latest/javadoc/org/apache/geode/cache/RegionShortcut.html	The `RegionShortcut` for this Region. Allows easy initialization of the Region based on pre-defined defaults.
statistics	boolean, default:false	Determines 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.	Expected value type.

CacheListeners

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

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

```
<gfe:replicated-region id="regionWithListeners">
  <gfe:cache-listener>
    <!-- nested CacheListener bean reference -->
    <ref bean="myListener"/>
    <!-- nested CacheListener bean definition -->
    <bean class="org.example.app.geode.cache.AnotherSimpleCacheListener"/>
  </gfe:cache-listener>

  <bean id="myListener" class="org.example.app.geode.cache.SimpleCacheListener"/>
</gfe:replicated-region>
```

The following example uses an alternate form of the `cache-listener` element with the `ref` attribute. This allows for more concise configuration when defining a single `CacheListener`. Note, 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 the `cache-listener` element is illegal. The two options are mutually exclusive and using both in the same element will result in an exception.

```
<beans>
  <gfe:replicated-region id="exampleReplicateRegionWithCacheListener">
    <gfe:cache-listener ref="myListener"/>
  </gfe:replicated-region>

  <bean id="myListener" class="example.CacheListener"/>
</beans>
```

NOTE	*Bean Reference Conventions* The `cache-listener` element is an example of a common pattern used in the namespace anywhere Geode 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.

CacheLoaders and CacheWriters

Similar to `cache-listener`, the namespace provides `cache-loader` and `cache-writer` elements to register these Geode components respectively for a Region.

A `CacheLoader` is invoked on a cache miss to allow an entry to be loaded from an external data source, such as a database. A `CacheWriter` is invoked before an entry is created or updated, intended for synchronizing to an external data source. The difference is Geode only supports at most a single instance `CacheLoader` and `CacheWriter` per Region. However, either declaration style may be used.

Example:

```
<beans>
  <gfe:replicated-region id="exampleReplicateRegionWithCacheLoaderAndCacheWriter">
    <gfe:cache-loader ref="myLoader"/>
    <gfe:cache-writer>
      <bean class="example.CacheWriter"/>
    </gfe:cache-writer>
  </gfe:replicated-region>

  <bean id="myLoader" class="example.CacheLoader">
    <property name="dataSource" ref="mySqlDataSource"/>
  </bean>

  <!-- DataSource bean definition -->
</beans>
```

See `CacheLoader` and `CacheWriter` in the Apache Geode documentation for more details.

5.5.4. Compression

Geode Regions may also be compressed in order to reduce JVM memory consumption and pressure to possibly avoid stop the world GCs. When you enable compression for a Region, all values stored in the Region, in-memory are compressed while keys and indexes remain uncompressed. New values are compressed when put into Region and all values are decompressed automatically when read back from the Region. Values are not compressed when persisted to disk or when sent over the wire to other peer members or clients.

Example:

```
<beans>
  <gfe:replicated-region id="exampleReplicateRegionWithCompression">
    <gfe:compressor>
      <bean class="org.apache.geode.compression.SnappyCompressor"/>
    </gfe:compressor>
  </gfe:replicated-region>
</beans>
```

Please refer to Apache Geode's documentation for more information on Region Compression.

5.5.5. Subregions

Spring Data Geode also supports Subregions, allowing Regions to be arranged in a hierarchical relationship.

For example, Geode 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 Geode 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:

```
<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 application beans that need them, such as `GemfireTemplate`. The full path should also be used in OQL query strings.

5.5.6. Region Templates

Spring Data Geode also supports 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* application context.

Spring Data Geode includes 5 Region template tags in namespace:

Table 2. Region Template Tags

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

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

Here is an example of 1 possible configuration...

```
<beans>
  <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" initial-capacity="51" load-factor="
0.85" persistent="false" statistics="true"
      key-constraint="java.lang.Long" value-constraint="java.lang.String">
    <gfe:cache-listener>
      <bean class="example.CacheListenerOne"/>
      <bean class="example.CacheListenerTwo"/>
    </gfe:cache-listener>
    <gfe:entry-ttl timeout="600" action="DESTROY"/>
    <gfe:entry-tti timeout="300 action="INVLIDATE"/>
  </gfe:region-template>

  <gfe:region-template id="ExtendedRegionTemplate" template="BaseRegionTemplate" 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:async-event-queue-ref bean="AEQ"/>
  </gfe:region-template>

  <gfe:partitioned-region-template id="PartitionRegionTemplate" template=
"ExtendedRegionTemplate"
      copies="1" load-factor="0.70" local-max-memory="1024" total-max-memory="16384"
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" persistent="true" total-buckets="91"/>
</beans>
```

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

How Templating Works

Spring Data Geode 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 must be defined 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.

Caution concerning Regions, Subregions and Lookups

Previously, one of the underlying properties of the `replicated-region`, `partitioned-region`, `local-region` and `client-region` elements in the *Spring Data Geode* XML namespace was to perform a lookup first before attempting to create a Region. This was done in case the Region already existed, which would be the case if the Region was defined in an imported Geode native `cache.xml` configuration file. Therefore, the lookup was performed first to avoid any errors. This was by design and subject to change.

This behavior has been altered and the default behavior is now to create the Region first. If the Region already exists, then the creation logic fails-fast and an appropriate exception is thrown. However, much like the `CREATE TABLE IF NOT EXISTS …` DDL syntax, the *Spring Data Geode* `<*-region>` namespace elements now includes a `ignore-if-exists` attribute, which re-instates the old behavior by performing a lookup of an existing Region identified by name, first. If an existing Region by name is found and `ignore-if-exists` is set to `true`, then the Region bean definition defined in *Spring* config is ignored.

WARNING	The *Spring* team highly recommends that the `replicated-region`, `partitioned-region`, `local-region` and `client-region` namespace elements be strictly used for defining new Regions only. One problem that could arise if the Regions defined by these elements already existed and the Region elements performed a lookup first is if the developer defined different Region semantics and behaviors for eviction, expiration, subscription, etc in his/her application config, then the Region definition may not match and could exhibit contrary behaviors to those required by the application. Even worse, the application developer may want to define the Region as a distributed Region (e.g. PARTITION) but in fact the existing Region definition is LOCAL.

IMPORTANT	Recommended Practice - Only use `replicated-region`, `partitioned-region`, `local-region` and `client-region` namespace elements to define new Regions.

Consider the following native Geode `cache.xml` configuration file…

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

  <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 may have defined an application 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` Region in our application DAO. As such, it is not uncommon for a developer to define beans for all or even 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/geode
http://www.springframework.org/schema/gemfire/spring-geode.xsd
">

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

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

</beans>
```

The `Customers/Accounts` and `Customers/Accounts/Orders` Regions are referenced as beans in the *Spring* application 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 (i.e. `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 top-level `replicated-region` element along with the `ignore-if-exists` attribute set to perform a lookup first, as in...

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

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

Geode is flexible in referencing both parent Regions and Subregions with or without the leading forward slash. For example, the parent can be referenced as "/Customers" or "Customers" and the child as "/Customers/Accounts" or just "Customers/Accounts". However, _Spring Data _Geode is very specific when it comes to naming beans after Regions, typically always using the forward slash (/) to represent Subregions (e.g. "/Customers/Accounts").

Therefore, it is recommended that users either use the nested `lookup-region` syntax as shown 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 the native Geode `cache.xml` configuration file as `REPLICATES` and will exist by the time the cache is initialized, or once the `<gfe:cache>` bean is processed.

5.5.7. Data Eviction (with Overflow)

Based on various constraints, each Region can have an eviction policy in place for evicting data from memory. Currently, in Geode, eviction applies to the *Least Recently Used* entry (also known as LRU). Evicted entries are either destroyed or paged to disk (referred to as **overflow** to disk).

Spring Data Geode supports all eviction policies (entry count, memory and heap usage) for PARTITION Regions, REPLICATE Regions and client, local Regions using the nested `eviction` element.

For example, to configure a PARTITION Region to overflow to disk if the memory size exceeds more than 512 MB, a developer would specify the following configuration:

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

IMPORTANT	Replicas cannot use `local destroy` eviction since that would invalidate them. See the Geode 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, please refer to the Geode documentation on Eviction.

5.5.8. Data Expiration

Apache Geode 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 the entry count or heap/memory usage.

Once an entry expires it may no longer be accessed from the cache.

Geode supports the following Expiration types:

- **Time-to-Live (TTL)** - The amount of time in seconds that an 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 that an 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 Geode* provides `<region-ttl>`, `<region-tti>`, `<entry-ttl>` and `<entry-tti>` Region child elements to specify timeout values and expiration actions.

For example:

```
<gfe:partitioned-region id="examplePartitionRegionWithExpiration">
  <gfe:region-ttl timeout="30000" action="INVALIDATE"/>
  <gfe:entry-tti timeout="600" action="LOCAL_DESTROY"/>
</gfe:replicated-region>
```

For a detailed description of expiration policies, please refer to the Geode documentation on Expiration.

Annotation-based Data Expiration

With *Spring Data Geode*, a developer 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 class SessionBasedApplicationDomainObject {
  ...
}
```

In addition, a developer may also specify Expiration type specific settings on Region Entries using `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` annotations 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 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 Geode's* Expiration annotation support. However, Apache Geode and *Spring Data Geode* 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 Geode's `@Expiration` annotation support is implemented with Geode's `CustomExpiry` interface. Refer to Geode's documentation on Configuring Data Expiration for more details

The *Spring Data Geode* `AnnotationBasedExpiration` class (and `CustomExpiry` implementation) is 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 Geode* to configure specific Geode 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 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 in the `AnnotationBasedExpiration` class, like so...

```
<bean id="ttlExpiration" class=
"org.springframework.data.gemfire.expiration.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 Geode's* `@Expiration` annotations: `@Expiration`, `@IdleTimeoutExpiration` and/or `@TimeToLiveExpiration`
3. (optional) In cases where particular application domain objects have not been annotated with *Spring Data Geode's* `@Expiration` annotations at all, but the Geode 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=
"org.apache.geode.cache.ExpirationAttributes">
    <constructor-arg value="600"/>
    <constructor-arg value="#{T(org.apache.geode.cache.ExpirationAction).DESTROY}"/>
</bean>

<bean id="ttiExpiration" class=
"org.springframework.data.gemfire.expiration.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 *Spring Data Geode's* `@Expiration` annotations use a 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 Geode'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 = "${geode.region.entry.expiration.ttl.timeout}"
    action = "${geode.region.entry.expiration.ttl.action}")
public class ExampleApplicationDomainObject {
  ...
}
```

Then, in your *Spring* XML config or in JavaConfig, you would declare the following beans…

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

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

Then, on your application domain object...

```
@TimeToLiveExpiration(timeout =
"@expirationSettings['geode.region.entry.expiration.ttl.timeout']"
    action = "@expirationSetting['geode.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`) uses SpEL to base the action value on the actual Expiration action enumerated type leading to more quickly identified failures if the types ever change.

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

5.5.9. Data Persistence

Regions can be persistent. Geode ensures that all the data you put into a Region that is configured for persistence will be written to disk in a way that is recoverable the next time you recreate the Region. This allows data to be recovered after machine or process failure, or even after an orderly shutdown and subsequent restart of the Geode data node.

To enable persistence with *Spring Data Geode*, simply set the `persistent` attribute to `true` on any of the `<*-region>` elements. For example…

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

Persistence may also be configured using the `data-policy` attribute; set the attribute's value to one of Geode's DataPolicy settings. For example…

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

The `DataPolicy` must match the Region type and must also agree with the `persistent` attribute if also explicitly set. An initialization exception will be thrown if the `persistent` attribute is set to `false` yet a persistent `DataPolicy` was specified (e.g. PERSISTENT_REPLICATE, PERSISTENT_PARTITION).

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="yetAnotherExamplePersistentPartitionRegion" persistent=
"true"
    disk-store-ref="myDiskStore" disk-synchronous="true"/>
```

This is discussed further in Configuring a DiskStore

5.5.10. Subscription Policy

Geode allows configuration of peer-to-peer (P2P) event messaging to control the entry events that the Region will receive. *Spring Data Geode* provides the `<gfe:subscription/>` sub-element to set the subscription policy on REPLICATE and PARTITION Regions to either `ALL` or `CACHE_CONTENT`.

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

5.5.11. Local Region

Spring Data Geode 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 apply.

A minimal declaration looks as follows (again, the example relies on the *Spring Data Geode* namespace naming conventions to wire the cache):

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

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

5.5.12. Replicated Region

One of the common Region types is a **REPLICATE** Region or **replica**. In short, when a Region is configured to be a REPLICATE, every member that hosts the Region stores a copy of the Region's entries locally. Any update to a REPLICATE 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 Geode offers a `replicated-region` element. A minimal declaration looks as follows. All common configuration options are available for REPLICATE Regions.

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

Refer to Geode's documentation on Distributed and Replicated Regions for more details.

5.5.13. Partitioned Region

Another Region type supported out-of-the-box by the *Spring Data Geode* namespace is the PARTITION Region.

To quote the Geode 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. Geode 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 partition 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:

```
<gfe:partitioned-region id="examplePartitionRegion" copies="2" total-buckets="17">
  <gfe:partition-resolver>
    <bean class="example.PartitionResolver"/>
  </gfe:partition-resolver>
</gfe:partitioned-region>
```

Refer to Geode's documentation on Partitioned Regions for more details.

Partitioned Region Attributes

The following table offers a quick overview of configuration options specific to PARTITION Regions. These are in addition to the common Region configuration options described above.

Table 3. partitioned-region attributes

Name	Values	Description
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 PARTITION Region with which this newly created PARTITION Region is collocated.
local-max-memory	**positive integer**	The maximum amount of memory in megabytes used by the Region in **this** process.
total-max-memory	**any integer value**	The maximum amount of memory in megabytes used by the Region in **all** processes.
partition-listener	**bean name**	The name of the `PartitionListener` used by this Region, for handling partition events.
partition-resolver	**bean name**	The name of the `PartitionResolver` used by this Region, for custom partitioning.
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.

Name	Values	Description
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

Apache Geode supports various deployment topologies for managing and distributing data. Geode topologies is outside the scope of this documentation. However, to quickly recap, Geode's supported topologies can be classified in short as: *peer-to-peer* (p2p), *client-server*, and *wide area network* (WAN). In the last two configurations, it is common to declare **client** Regions which connect to a cache server.

Spring Data Geode offers dedicated support for such configuration through client-cache, `client-region` and `pool` elements. As the names imply, the former defines a client Region while the latter defines a Pool of connections to be used/shared by the various client Regions.

Below is a typical client Region configuration:

```
<bean id="myListener" class="example.CacheListener"/>

<!-- client Region using the default SDG gemfirePool Pool -->
<gfe:client-region id="Example">
  <gfe:cache-listener ref="myListener"/>
</gfe:client-region>

<!-- client Region using its own dedicated Pool -->
<gfe:client-region id="AnotherExample" pool-name="myPool">
  <gfe:cache-listener ref="myListener"/>
</gfe:client-region>

<!-- Pool definition -->
<gfe:pool id="myPool" subscription-enabled="true">
  <gfe:locator host="remoteHost" port="12345"/>
</gfe:pool>
```

As with the other Region types, `client-region` supports `CacheListener``s` as well as a `CacheLoader` and `CacheWriter`. It also requires a connection `Pool` for connecting to either a set of Locators or Servers. Each client Region can have its own Pool or they can share the same one.

NOTE	In the above example, the Pool is configured with `locator`. A Locator is a separate process used to discover cache servers and peer data members 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 Geode* schema (Spring Data Geode Schema) and Geode's documentation on Client/Server Configuration.

Client Interests

To minimize network traffic, each client can separately define its own 'interests' policies, indicating to Geode the data it actually requires. In *Spring Data Geode*, 'interests' can be defined for each client Region separately. Both Key-based and Regular Expression-based interest types are supported.

For example:

```
<gfe:client-region id="Example" pool-name="myPool">
    <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. The same can be accomplished using a regex of `".\*"`.

The `<gfe:*-interest>` *Key* and *Regular Expression* elements support 3 attributes: `durable`, `receive-values` and `result-policy`.

`durable` indicates whether the 'interest' policy and subscription queue created for the client when the client connects to 1 or more servers in the cluster is maintained across client sessions. If the client goes away and comes back, a "durable" subscription queue on the server(s) for the client is maintained while the client is disconnected, and when the client reconnects, the client will receive any events that occurred while the client was disconnected from the servers(s) in the cluster.

A subscription queue on the servers in the cluster is maintained for each `Pool` of connections defined in the client where subscription has also been "enabled" for that `Pool`. The subscription queue is used to store, and possibly conflate, events sent to the client. If the subscription queue is durable, it persists between client sessions (i.e. connections), potentially up to a specified timeout (if the client does not return within a given time frame in order to reduce resource consumption on servers in the cluster). If the subscription queue is not "durable", then it will be destroyed when the client disconnects. All you need to decide is, for your application use case, is it important for the cache client to receive events while it is disconnected, or is it only important for the application (cache client) to receive the "latest" events after it reconnects.

The `receive-values` attribute indicates whether or not the entry values are received for create and update events. If **true**, values are received; if **false**, only invalidation events are received.

And finally, the 'result-policy` is an enumeration of: `KEYS`, `KEYS_VALUE` and `NONE`. The default is `KEYS_VALUES`. The `result-policy` controls the initial dump when the client first connects to initialize the local cache, essentially seeding the client with events for all the entries that match the interest policy.

Client-side interests registration does not do much good without enabling subscription on the `Pool` as mentioned above. In fact, it is an error to attempt interests registration without subscription enabled. To do so, you simply...

```
<gfe:pool ... subscription-enabled="true">
  ...
</gfe:pool>
```

In addition to `subscription-enabled`, can you also set `subscription-ack-interval`, `subscription-message-tracking-timeout` and `subscription-redundancy`. `subscription-redundancy` is used to control how many copies of the subscription queue should be maintained by the servers in the cluster. If redundancy is greater than 1, and the "primary" subscription queue (i.e. server) goes down, then a "secondary" subscription queue will take over, keeping the client from missing events in a HA scenario.

In addition to the `Pool` settings, the server-side Regions use an additional attribute, `enable-subscription-conflation`, to control the conflation of events that will be sent to the clients. This can also help further minimize network traffic and is useful in situations where the application only cares about the latest value of an entry. However, in cases where the application is keeping a time series of events that occurred, conflation is going to hinder that use case. The default value is **false**. An example Region configuration on the server for which the client contains a corresponding client [CACHING_]PROXY Region with interests in Keys in this server Region, would look like...

```
<gfe:partitioned-region name="ServerSideRegion" enable-subscription-conflation="true">
  ...
</gfe:partitioned-region>
```

To control the amount of time in seconds that "durable" subscription queue is maintained after a client is disconnected from the server(s) in the cluster, set the `durable-client-timeout` attribute on the `<gfe:client-cache>` element like so...

```
<gfe:client-cache durable-client-timeout="600">
  ...
</gfe:client-cache>
```

A full, in-depth discussion of how client interests work and capabilities is beyond the scope of this document.

Please refer to Apache Geode's documentation on Client-to-Server Event Distribution for more details.

5.5.15. JSON Support

Apache Geode has support for caching JSON documents in Regions along with the ability to query stored JSON documents using the Geode OQL. JSON documents are stored internally as PdxInstance types using the JSONFormatter class to perform conversion to and from JSON documents (as a `String`).

Spring Data Geode provides the `<gfe-data:json-region-autoproxy/>` element to enable a AOP, *Spring* component to advise appropriate, proxied Region operations, which effectively encapsulates the `JSONFormatter`, thereby allowing your applications to work directly with JSON Strings.

In addition, Java objects written to JSON configured Regions will be automatically converted to JSON using Jackson's `ObjectMapper`. Reading these values back will be returned as a JSON String.

By default, `<gfe-data:json-region-autoproxy/>` performs the conversion for all Regions. To apply this feature to selected Regions, provide a comma delimited list of Region bean ids via the `region-refs` attribute. Other attributes include a `pretty-print` flag (defaults to **false**) and `convert-returned-collections`.

Also by default, the results of the `getAll()` and `values()` Region operations will be converted for configured Regions. This is done by creating a parallel data structure in local memory. This can incur significant overhead for large collections, so set the `convert-returned-collections` to **false** if you would like to disable automatic conversion for these Region operations.

NOTE	Certain Region operations, specifically those that use Geode'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 also not affected.

Example configuration:

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

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

5.6. Configuring an Index

Apache Geode allows Indexes (or Indices) to be created on Region data to improve the performance of OQL queries.

In *Spring Data Geode* (SDG), Indexes are declared with the `index` element:

```
<gfe:index id="myIndex" expression="someField" from="/SomeRegion" type="HASH"/>
```

In *Spring Data Geode's* XML schema (a.k.a. SDG namespace), `Index` bean declarations are not bound to a *Region*, unlike Geode's native `cache.xml`. Rather, they are top-level elements just like `<gfe:cache>`. This allows a developer to declare any number of Indexes on any *Region* whether they were just created or already exist, a significant improvement over Geode's native `cache.xml` format.

An `Index` must have a name. A developer may give the `Index` an explicit name using the `name` attribute, otherwise the *bean name* (i.e. value of the `id` attribute) of the `Index` bean definition is used as the `Index` name.

The `expression` and `from` clause form the main components of an `Index`, identifying the data to index (i.e. the *Region* identified in the `from` clause) along with what criteria (i.e. `expression`) is used to index the data. The `expression` should be based on what application domain object fields are used in the predicate of application-defined OQL queries used to query and lookup the objects stored in the *Region*.

For example, if I have a `Customer` that has a `lastName` property...

```
@Region("Customers")
class Customer {

  @Id
  Long id;

  String lastName;
  String firstName;

  ...
}
```

And, I also have an application defined SD[G] *Repository* to query for `Customers`...

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

  Customer findByLastName(String lastName);

  ...
}
```

Then, the SD[G] *Repository* finder/query method would result in the following OQL statement being executed...

```
SELECT * FROM /Customers c WHERE c.lastName = '$1'
```

Therefore, I might want to create an `Index` like so...

```
<gfe:index id="myIndex" name="CustomersLastNameIndex" expression="lastName" from=
"/Customers" type="HASH"/>
```

The `from` clause must refer to a valid, existing *Region* and is how an `Index` gets applied to a *Region*. This is **not** *Sprig Data Geode* specific; this is a feature of Apache Geode.

The `Index type` maybe 1 of 3 enumerated values defined by *Spring Data Geode's* IndexType enumeration: `FUNCTIONAL`, `HASH` and `PRIMARY_KEY`.

Each of the enumerated values correspond to one of the QueryService `create[|Key|Hash]Index` methods invoked when the actual `Index` is to be created (or "defined"; more on "defining" Indexes below). For instance, if the `IndexType` is `PRIMARY_KEY`, then the QueryService.createKeyIndex(..) is invoked to create a `KEY Index`.

The default is `FUNCTIONAL` and results in one of the `QueryService.createIndex(..)` methods being invoked.

See the *Spring Data Geode* XML schema for a full set of options.

For more information on Indexing in Apache Geode, see Working with Indexes in Apache Geode's User Guide.

5.6.1. Defining Indexes

In addition to creating Indexes upfront as `Index` bean definitions are processed by *Spring Data Geode* on *Spring* container initialization, you may also **define** all of your application Indexes prior to creating them by using the `define` attribute, like so...

```
<gfe:index id="myDefinedIndex" expression="someField" from="/SomeRegion" define="true
"/>
```

When `define` is set to `true` (defaults to `false`), this will not actually create the `Index` right then and there. All "defined" Indexes are created all at once, when the *Spring* `ApplicationContext` is "refreshed", or, that is, when a `ContextRefreshedEvent` is published by the *Spring* container. *Spring Data Geode* registers itself as an `ApplicationListener` listening for the `ContextRefreshedEvent`. When fired, *Spring Data Geode* will call QueryService.createDefinedIndexes().

Defining Indexes and creating them all at once helps promote speed and efficiency when creating Indexes.

See Creating Multiple Indexes at Once for more details.

5.6.2. `IgnoreIfExists` and `Override`

Two *Spring Data Geode* `Index` configuration options warrant special mention here: `ignoreIfExists` and `override`.

These options correspond to the `ignore-if-exists` and `override` attributes on the `<gfe:index>`

element in *Spring Data Geode's* XML schema, respectively.

WARNING	Make sure you absolutely understand what you are doing before using either of these options. These options can affect the performance and/or resources (e.g. memory) consumed by your application at runtime. As such, both of these options are disabled (i.e. set to `false`) in SDG by default.

NOTE	These options are only available in *Spring Data Geode* and exist to workaround known limitations with Apache Geode; there are no equivalent options or functionality available in Geode itself.

Each option significantly differs in behavior and entirely depends on the type of Geode `Index` *Exception* thrown. This also means that neither option has any effect if a Geode Index-type *Exception* is **not** thrown. These options are meant to specifically handle Geode `IndexExistsExceptions` and `IndexNameConflictExceptions`, which can occur for various, sometimes obscure reasons. But, in general…

- An IndexExistsException is thrown when there exists another `Index` with the same definition but different name when attempting to create an `Index`.
- An IndexNameConflictException is thrown when there exists another `Index` with the same name but possibly different definition when attempting to create an `Index`.

Spring Data Geode's default behavior is to ***fail-fast***, always! So, neither `Index` *Exception* will be "handled" by default; these `Index` *Exceptions* are simply wrapped in a SDG `GemfireIndexException` and rethrown. If you wish for *Spring Data Geode* to handle them for you, then you can set either of these `Index` bean definition options.

`IgnoreIfExists` always takes **precedence** over `Override`, primarily because it uses less resources given it returns the "existing" `Index` in both exceptional cases.

`IgnoreIfExists` Behavior

When an `IndexExistsException` is thrown and `ignoreIfExists` is set to `true` (or `<gfe:index ignore-if-exists="true">`), then the `Index` that would have been created by this `Index` bean definition / declaration will be "**ignored**", and the "existing" `Index` will be returned.

There is very little consequence in returning the "existing" `Index` since the `Index` "definition" is the same, as deemed by Geode itself, **not** SDG.

However, this also means that **no** `Index` with the "name" specified in your `Index` bean definition / declaration will "actually" exist from Geode's perspective either (i.e. with QueryService.getIndexes()). Therefore, you should be careful when writing OQL query statements that use *Query Hints*, especially *Hints* that refer to the application `Index` being "**ignored**". Those *Query Hints* will need to be changed.

Now, when an `IndexNameConflictException` is thrown and `ignoreIfExists` is set to `true` (or `<gfe:index ignore-if-exists="true">`), then the `Index` that would have been created by this `Index` bean definition / declaration will also be "**ignored**", and the "existing" Index will be returned, just like when an `IndexExistsException` is thrown.

However, there is more risk in returning the "existing" `Index` and "**ignoring**" the application's definition of the `Index` when an `IndexNameConflictException` is thrown since, for a `IndexNameConflictException`, while the "names" of the conflicting Indexes are the same, the "definitions" could very well be different! This obviously could have implications for OQL queries specific to the application, where you would presume the Indexes were defined specifically with the application data access patterns and queries in mind. However, if like named Indexes differ in definition, this might not be the case. So, make sure you verify.

NOTE	SDG makes a best effort to inform the user when the `Index` being ignored is significantly different in its definition from the "existing" `Index`. However, in order for SDG to accomplish this, it must be able to "find" the existing `Index`, which is looked up using the Geode API (the only means available).

`Override` Behavior

When an `IndexExistsException` is thrown and `override` is set to `true` (or `<gfe:index override="true">`), then the `Index` is effectively "*renamed*". Remember, `IndexExistsExceptions` are thrown when multiple Indexes exist, all having the same "definition" but different "names".

Spring Data Geode can only accomplish this using Geode's API, by first "*removing*" the "existing" `Index` and then "*recreating*" the `Index` with the **new** name. It is possible that either the remove or subsequent create invocation could fail. There is no way to execute both actions atomically and rollback this joint operation if either fails.

However, if it succeeds, then you have the same problem as before with the "*ignoreIfExists*" option. Any existing OQL query statement using "*Query Hints*" referring to the old `Index` by name must be changed.

Now, when an `IndexNameConflictException` is thrown and `override` is set to `true` (or `<gfe:index override="true">`), then potentially the "existing" `Index` will be "*re-defined*". I say "potentially", because it is possible for the "like-named", "existing" `Index` to have exactly the same definition and name when an `IndexNameConflictException` is thrown.

If so, SDG is **smart** and will just return the "existing" Index as is, even on `override`. There is no harm in this since both the "name" and the "definition" are exactly the same. Of course, SDG can only accomplish this when SDG is able to "find" the "existing" `Index`, which is dependent on Geode's APIs. If it cannot find it, nothing happens and a SDG `GemfireIndexException` is thrown wrapping the `IndexNameConflictException`.

However, when the "definition" of the "existing" `Index` is different, then SDG will attempt to "*recreate*" the `Index` using the `Index` definition specified in the `Index` bean definition /declaration. Make sure this is what you want and make sure the `Index` definition matches your expectations and application requirements.

How does `IndexNameConflictExceptions` actually happen?

It is probably not all that uncommon for `IndexExistsExceptions` to be thrown, especially when multiple configuration sources are used to configure Geode (e.g. *Spring Data Geode*, Geode *Cluster Config*, maybe Geode native `cache.xml`, the API, etc, etc). You should definitely prefer 1 configuration

method here and stick with it.

However, when does an `IndexNameConflictException` *get thrown?*

One particular case is an `Index` defined on a `PARTITION` *Region* (PR). When an `Index` is defined on a `PARTITION` *Region* (e.g. "X"), Geode distributes the `Index` definition (and name) to other peer members in the cluster that also host the same `PARTITION` *Region* (i.e. "X"). The distribution of this `Index` definition to and subsequent creation of this `Index` by peer members on a "need-to-know" basis (i.e. those hosting the same PR) is performed asynchronously.

During this window of time, it is possible that these "pending" PR `Indexes` will not be identifiable by Geode, such as with a call to QueryService.getIndexes() or with QueryService.getIndexes(:Region).

As such, the only way for SDG or other Geode cache client applications (not involving *Spring*) to know for sure, is to just attempt to create the `Index`. If it fails with either an `IndexNameConflictException`, or even an `IndexExistsException`, then you will know. This is because the `QueryService` `Index` creation waits on "pending" `Index` definitions, where as the other Geode API calls do not.

In any case, SDG makes a best effort and attempts to inform the user what has or is happening along with the corrective action. Given all Geode `QueryService.createIndex(..)` methods are synchronous, "blocking" operations, then the state of Geode should be consistent and accessible after either of these Index-type *Exceptions* are thrown, in which case, SDG can inspect the state of the system and respond/act accordingly, based on the user's desired configuration.

In all other cases, SDG will simply ***fail-fast***!

5.7. Configuring a DiskStore

Spring Data Geode supports `DiskStore` configuration via the `disk-store` element.

For example:

```
<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>
```

`DiskStores` are used by Regions for file system persistent backup and overflow of evicted entries as well as persistent backup of WAN Gateways. Multiple Geode components may share the same `DiskStore`. Additionally, multiple file system directories may be defined for a single `DiskStore`.

Please refer to Apache Geode's documentation for a complete explanation of the configuration options.

5.8. Configuring the Snapshot Service

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

As the Apache Geode documentation describes, snapshots allow you to save and subsequently reload the cached data later, which can be useful for moving data between environments, such as 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 Geode's* Snapshot Service support with Spring's bean definition profiles to load snapshot data specific to the environment as necessary.

Spring Data Geode's support for Apache Geode's Snapshot Service begins with the `<gfe-data:snapshot-service>` element from the `<gfe-data>` namespace.

For example, I might want to define Cache-wide snapshots to be loaded as well as saved using a couple snapshot imports and a 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 Geode* application, JVM process's working directory.

This is a pretty simple example and the Snapshot Service defined in this case refers to the Geode `Cache` with the default name of `gemfireCache` (as described in Configuring a Cache). If you name your cache bean definition something other than the default, 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 particular Geode 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="/absolute/path/to/export/example.snapshot/>
</gfe-data:snapshot-service>
```

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

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

TIP *Spring Data Geode* includes a `suppress-import-on-init` attribute on the `<gfe-data:snapshot-service>` element to suppress the configured Snapshot Service from trying to import data into the Cache or 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 Snapshot Service (i.e. 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 Geode `Cache`.

However, *Spring Data Geode* 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.

Therefore, *Spring Data Geode* enables the developer to specify a JAR or ZIP file on import for a `Cache`-based Snapshot Service 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 Geode 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 Apache Geode'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 Geode 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="example.geode.snapshot.filter.ActiveUsersFilter/>

<gfe-data:snapshot-service id="adminsSnapshotService" region-ref="Admins">
  <gfe-data:snapshot-import location="/path/to/import/users.snapshot">
    <bean class="example.geode.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="example.geode.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 Geode* provided class. This class implements Geode's SnapshotFilter interface as well as the Composite software design pattern.

In a nutshell, the Composite software design pattern allows developers to compose multiple objects of the same type and treat the aggregate as single instance of the object type, a very powerful and useful abstraction.

`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 take a list of `SnapshotFilters`.

In this case, the developer 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>
```

The developer 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="example.geode.snapshot.filter.CovertUsersFilter"/>
    </list>
  </constructor-arg>
</bean>
```

5.8.3. Snapshot Events

By default, *Spring Data Geode* uses Apache Geode'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 *Spring* application.

For this purpose, *Spring Data Geode* 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 Geode Cache, or individual Geode Regions. The constructors in these 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* `ApplicationContext`. 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 the `region-ref` attribute) 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 Snapshot Service 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 the Function Service

Spring Data Geode provides annotation support for implementing and registering Apache Geode Functions.

Spring Data Geode also provides namespace support for registering Apache Geode Functions for remote Function execution.

Please refer to Apache Geode' documentation for more information on the Function execution framework.

Geode Functions are declared as *Spring* beans and must implement the `org.apache.geode.cache.execute.Function` interface or extend `org.apache.geode.cache.execute.FunctionAdapter`.

The namespace uses a familiar pattern to declare functions:

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

<bean id="function2" class="example.FunctionTwo"/>
```

5.10. Configuring WAN Gateways

WAN Gateways provide a way to synchronize Apache Geode Distributed Systems across geographic areas. *Spring Data Geode* provides namespace support for configuring WAN Gateways as illustrated in the following examples.

5.10.1. WAN Configuration in GemFire 7.0

In the example below, `GatewaySenders` are configured for a PARTITION Region by adding child elements to the Region (`gateway-sender` and `gateway-sender-ref`).

A `GatewaySender` may register `EventFilters` 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-with-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="example.AsyncEventListener"/>
    </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 `EventFilters` and `TransportFilters`.

```
<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 Apache Geode documentation for a detailed explanation of all the configuration

options.

Chapter 6. Bootstrapping Apache Geode using Spring Annotations

Spring Data Geode (SDG) 2.0 introduces a **new** Annotation-based configuration model to bootstrap Apache Geode with the Spring container.

The primary motivation for introducing an Annotation-based approach to the configuration of Apache Geode in a Spring context is to enable application developers to ***get up and running*** *as* ***quickly*** *and as* ***easily*** *as possible*.

6.1. Introduction

Apache Geode can be very difficult to setup and use successfully given all the configuration properties, configuration options (cache.xml, *Gfsh* + *Cluster Configuration*, Spring XML/Java-based configuration) along with different supported *topologies* (client/server, P2P, WAN) and *Distributed System Design Patterns* (e.g. shared-nothing architecture). The Annotation-based configuration model aims to simplify all this plus more.

The Annotation-based configuration model is an alternative to XML-based configuration using *Spring Data Geode's* XML Namespace. With XML, an application developer would use the `spring-gemfire` (`gfe`) schema for configuration and the `spring-data-gemfire` (`gfe-data`) schema for data access related concerns. See Bootstrapping Apache Geode with the Spring Container for more details.

NOTE	As of SDG 2.0, the new Annotation-based configuration model does not yet have configuration support for Apache Geode's WAN components and topology.

Like *Spring Boot*, *Spring Data Geode's* Annotation-based configuration model was designed as an opinionated, *convention over configuration* approach for using Apache Geode. Indeed, the Annotation-based configuration model was inspired by *Spring Boot* as well as several other Spring and *Spring Data* projects.

By following convention, all Annotations provide reasonable and sensible defaults for all the attributes out-of-the-box. The default value for a given Annotation attribute directly corresponds to the default value provided in Apache Geode for the same configuration property or setting.

The intention is to let an application developer enable an Apache Geode feature or an embedded service by simply declaring the Annotation on his/her Spring `@Configuration` or `@SpringBootApplcation` class without needing to unnecessarily configure a large number of attributes or properties just to use the feature.

Again, *getting up and running as quickly and as easily as possible* is the primary objective.

However, the option to customize the configuration meta-data and behavior of Apache Geode is there should an application developer need it and *Spring Data Geode's* Annotation-based configuration will quietly back away. The application developer simply just needs to specify the configuration attributes s/he wishes to adjust. And, as we will see below, there are several ways to

configure an Apache Geode feature or embedded service using Annotations.

All the **new** SDG Annotations can be found in the `org.springframework.data.gemfire.config.annotation` package.

6.2. Bootstrapping Apache Geode applications with Spring

Like all *Spring Boot* applications that begin by annotating the application class with `@SpringBootApplication`, a *Spring Boot* application can easily become an Apache Geode cache application simply by declaring 1 of 3 main Annotations:

1. `@ClientCacheApplication`
2. `@PeerCacheApplication`
3. `@CacheServerApplication`

These 3 Annotations are the Spring/Apache Geode application developer's starting point.

To realize the intent behind these Annotations, a user must understand that there are 2 types of cache instances that can be created with Apache Geode: a *client* or a *peer*.

A *Spring Boot* application can be configured as an Apache Geode cache client (i.e. with an instance of `ClientCache`), which communicates with an existing, standalone cluster of Apache Geode servers used to manage the application's data. The *client/server* topology is the most typical system architecture employed when using Apache Geode and the user can make her *Spring Boot* application a cache client simply by annotating it with `@ClientCacheApplication`.

Alternatively, a *Spring Boot* application may be a peer member of an Apache Geode cluster. That is, the application itself is just another server in the cluster of servers that will manage data. The application creates an "embedded" peer `Cache` instance when a developer annotates his/her application class with `@PeerCacheApplication`.

By extension, the application may also serve as a `CacheServer` for cache clients, allowing clients to connect and perform data access operations on the server. This is accomplished by annotating the application class with `@CacheServerApplication` in place of `@PeerCacheApplication`, which will create a peer `Cache` instance along with the `CacheServer`.

NOTE

An Apache Geode Server is not necessarily a "*Cache Server*" by default. That is, it is not necessarily setup to service cache clients just because it is a "server". A Geode Server can just be a peer member/data node of the cluster that manages data without servicing any clients while other peer members in the cluster are setup to service clients in addition to managing data. It also possible to setup certain peer members as non-data node, data accessors that can service clients as `CacheServers` as well, but is well beyond the scope of this document.

By way of example, if I wanted to create a *Spring Boot*, Apache Geode cache client application, I would start with…

Spring-based Apache Geode `ClientCache` application

```
@SpringBootApplication
@ClientCacheApplication
class ClientApplication { .. }
```

And, if I wanted to create a *Spring Boot* application with an embedded peer `Cache` instance, where my application will be a server and peer member of a cluster, or distributed system formed by Apache Geode, then I would start with...

Spring-based Apache Geode embedded peer `Cache` application

```
@SpringBootApplication
@PeerCacheApplication
class ServerApplication { .. }
```

Alternatively, a user may use the `@CacheServerApplication` annotation in place of `@PeerCacheApplication`, which will create both an "embedded" peer `Cache` instance along with a `CacheServer` running on "*localhost*", listening on the default cache server port, **40404**...

Spring-based Apache Geode embedded `CacheServer` Application

```
@SpringBootApplication
@CacheServerApplication
class ServerApplication { .. }
```

6.3. Going in-detail on *client/server* applications

There are multiple ways that a client can connect to and communicate with servers in a Geode cluster. The most common and recommended approach is to use Apache Geode Locators.

NOTE A cache client can connect to 1 or more Locators in the Geode cluster instead of directly to a `CacheServer`. The advantage of using Locators over direct `CacheServer` connections is that Locators provide meta-data about the cluster to which clients are connected. This meta-data includes information like which servers have the least amount of load, or which servers contain the data of interests to the client. A Locator also provides fail-over capabilities in case a `CacheServer` goes down. By enabling the PR single-hop capability in the client `Pool`, the client is routed directly to the server containing the data the client needs access to.

NOTE Locators are also peer members in a cluster. Locators actually constitute what makes up a cluster of Geode nodes; i.e. all nodes connected by a Locator make up a cluster of peers and new members use Locators to join a cluster and find other members.

Since Apache Geode sets up a "DEFAULT" `Pool` connected to a `CacheServer` running on "*localhost*", listening on port **40404** by default when a `ClientCache` instance is created, there is nothing special a

user need do to utilize the *client/server* topology. Simply annotate your server-side *Spring Boot* application with `@CacheServerApplication` and your client-side *Spring Boot* application with `@ClientCacheApplication` and you are all set.

You can even start your servers using *Gfsh's* `start server` command if you prefer. Your *Spring Boot* `@ClientCacheApplication` will still connect to the server regardless of how it is started. Although, we think you will prefer to configure and start your servers using the *Spring Data Geode* approach, with Annotations.

However, as an application developer, you will no doubt want to customize the "DEFAULT" `Pool` setup by Apache Geode to possibly connect to 1 or more Locators, for instance...

Spring-based Apache Geode `ClientCache` application using Locators

```
@SpringBootApplication
@ClientCacheApplication(locators = {
    @Locator(host = "boombox" port = 11235),
    @Locator(host = "skullbox", port = 12480)
})
class ClientApplication { .. }
```

Along with the `locators` attribute, the `@ClientCacheApplication` annotation has a `servers` attribute that can be used to specify 1 or more nested `@Server` annotations that enable the cache client to connect directly to 1 or more servers.

NOTE	You can only use either the `locators` or `servers` attribute, but not both, which is enforced by Apache Geode.

A user may also configure additional `Pools`, other than the "DEFAULT" `Pool` provided by Apache Geode when a `ClientCache` instance is created with the `@ClientCacheApplication` annotation, by using the `@EnablePool` and `@EnablePools` annotations.

NOTE	`@EnablePools` is a composite annotation that aggregates several nested `@EnablePool` annotations on a single class. Java 8 and earlier does not allow more than 1 annotation of the same type to be declared on a class.

Spring-based Apache Geode `ClientCache` application using multiple named `Pools`

```
@SpringBootApplication
@ClientCacheApplication(logLevel = "info")
@EnablePool(name = "VenusPool", servers = @Server(host = "venus", port = 48484),
    min-connections = 50, max-connections = 200, ping-internal = 15000,
    prSingleHopEnabled = true, readTimeout = 20000, retryAttempts = 1,
    subscription-enable = true)
@EnablePools(pools = {
    @EnablePool(name = "SaturnPool", locators = @Locator(host="skullbox", port=20668),
        subsription-enabled = true),
    @EnablePool(name = "NeptunePool", severs = {
            @Server(host = "saturn", port = 41414),
            @Server(host = "neptune", port = 42424)
        }, min-connections = 25))
})
class ClientApplication { .. }
```

The `name` attribute is the only required attribute of the `@EnablePool` annotation. As we will see below, the value of `name` corresponds to both the name of the `Pool` bean created in the Spring context as well as the name used to reference the corresponding configuration properties. It is also the name of the `Pool` registered and used in Apache Geode.

Similarly, on the server, a user can configure multiple `CacheServers` that a client can connect to...

Spring-based Apache Geode `CacheServer` application using multiple named `CacheServers`

```
@SpringBootApplication
@CacheSeverApplication(logLevel = "info", autoStartup = true, maxConnections = 100)
@EnableCacheServer(name = "Venus", autoStartup = true,
    hostnameForClients = "venus", port = 48484)
@EnableCacheServers(servers = {
    @EnableCacheServer(name = "Saturn", hostnameForClients = "saturn", port = 41414),
    @EnableCacheServer(name = "Neptune", hostnameForClients = "neptune", port = 42424)
})
class ServerApplication { .. }
```

NOTE Like `@EnablePools`, `@EnableCacheServers` is a composite annotation for aggregating multiple `@EnableCacheServer` annotations on a single class.

One thing an observant reader may have noticed is, in all cases, the user is specifying hard-coded values for hostnames, ports as well other configuration-oriented Annotation attributes. This is not ideal when a user's application gets promoted and deployed to different environments, such as from DEV to QA to STAGING to PROD.

How does an application developer handle dynamic configuration determined at runtime?

6.4. Runtime configuration using `Configurers`

Another goal when designing the Annotation-based configuration model was to preserve *Type-Safety* in the Annotation attributes. For example, if an attribute could be expressed as an `int`, like a port number, then the attribute's type should be an `int`.

Unfortunately, this is not conducive to dynamic and resolvable configuration at runtime.

One of the finer features of Spring is the ability to use *property placeholders* or *SpEL expressions* in properties or attributes of the configuration meta-data when configuring beans in a Spring context. Although, this would require all Annotation attributes be `Strings` thereby giving up *Type-Safety*; not acceptable!

So, *Spring Data Geode* borrows from another commonly used pattern in Spring, `Configurers`. Many different `Configurer` interfaces are provided out-of-the-box in Spring Web MVC, such as the `org.springframework.web.servlet.config.annotation.ContentNegotiationConfigurer`.

`Configurers` are a way to allow application developers to receive a callback and customize the configuration of a component on startup. The framework calls back to user-provided code to adjust the configuration at runtime. One of the more common uses of this pattern is to supply conditional configuration based on the application's runtime environment.

Spring Data Geode provides several `Configurer` callback interfaces to customize different aspects of Annotation-based configuration meta-data at runtime, before the *Sring* managed beans that the Annotations create are initialized:

- `ClientCacheConfigurer`
- `PeerCacheConfigurer`
- `CacheServerConfigurer`
- `ContinuousQueryListenerContainerConfigurer`
- `DiskStoreConfigurer`
- `IndexConfigurer`
- `PoolConfigurer`
- `RegionConfigurer`

For example, we can use the `CacheServerConfigurer` and `ClientCacheConfigurer` to customize the port numbers used by our `CacheServer` and `ClientCache` applications, respectively.

First, in our server application…

Customizing a Spring Boot `CacheServer` application with a `CacheServerConfigurer`

```
@SpringBootApplication
@CacheServerApplication(name = "SpringApplication", logLevel = "info")
class ServerApplication {

  @Bean
  CacheServerConfigurer cacheServerPortConfigurer(
          @Value("${geode.cache.server.host:localhost}") String cacheServerHost
          @Value("${geode.cache.server.port:40404}") int cacheServerPort) {

      return (beanName, cacheServerFactoryBean) -> {
          cacheServerFactoryBean.setBindAddress(cacheServerHost);
          cacheServerFactoryBean.setHostnameForClients(cacheServerHost);
          cacheServerFactoryBean.setPort(cacheServerPort);
      };
  }
}
```

Then, in our client application...

Customizing a Spring Boot `ClientCache` application with a `ClientCacheConfigurer`

```
@SpringBootApplication
@ClientCacheApplication(logLevel = "info")
class ClientApplication {

  @Bean
  ClientCacheConfigurer clientCachePoolPortConfigurer(
          @Value("${geode.cache.server.host:localhost}") String cacheServerHost
          @Value("${geode.cache.server.port:40404}") int cacheServerPort) {

      return (beanName, clientCacheFactoryBean) ->
          clientCacheFactoryBean.setServers(Collections.singletonList(
              new ConnectionEndpoint(cacheServerHost, cacheServerPort)));
  }
}
```

By using the provided `Configurers`, a user is able to receive a callback in order to further customize the configuration that is enabled by the associated Annotation.

In addition, when the `Configurer` is declared as a bean in the Spring context, the bean definition can take advantage of other Spring container features, such as *property placeholders*, or *SpEL expressions* in `@Value` annotations on factory method parameters, and so on.

All *Spring Data Geode*-provided `Configurers` take 2 bits of information in the callback: the name of the bean created in the Spring context by the Annotation along with a reference to the `FactoryBean` used by the Annotation to configure the Geode component (e.g. a `ClientCache` instance with SDG's `ClientCacheFactoryBean`).

NOTE	SDG `FactoryBeans` are part of the SDG public API and are what an application developer would use in Spring's Java-based container configuration if this **new** Annotation-based configuration model were not provided. Indeed, the Annotations themselves are using these very same `FactoryBeans` for their configuration.

Given a `Configurer` can be declared as a regular bean definition like any other, it is not difficult to imagine a user combining different Spring configuration options, such as the use of *Spring Profiles* with `Conditions` as well as other features to create even more sophisticated and flexible configuration.

However, `Configurers` are not the only option.

6.5. Runtime configuration using `Properties`

In addition to `Configurers`, each Annotation attribute in the Annotation-based configuration model is associated with a corresponding configuration *property*, prefixed with `spring.data.gemfire.`, that can be declared in *Spring Boot* `application.properties`.

Building on our examples above, the client's `application.properties` would define...

Client `application.properties`

```
spring.data.gemfire.cache.log-level=info
spring.data.gemfire.cache.pool.venus.servers=venus[48484]
spring.data.gemfire.cache.pool.venus.max-connections=200
spring.data.gemfire.cache.pool.venus.min-connections=50
spring.data.gemfire.cache.pool.venus.ping-interval=15000
spring.data.gemfire.cache.pool.venus.pr-single-hop-enabled=true
spring.data.gemfire.cache.pool.venus.read-timeout=20000
spring.data.gemfire.cache.pool.venus.subscription-enabled=true
spring.data.gemfire.cache.pool.saturn.locators=skullbox[20668]
spring.data.gemfire.cache.pool.saturn.subscription-enabled=true
spring.data.gemfire.cache.pool.neptune.servers=saturn[41414],neptune[42424]
spring.data.gemfire.cache.pool.neptune.min-connections=25
```

And, the server's application.properties would define...

Server `application.properties`

```
spring.data.gemfire.cache.log-level=info
spring.data.gemfire.cache.server.port=40404
spring.data.gemfire.cache.server.Venus.port=43434
spring.data.gemfire.cache.server.Saturn.port=41414
spring.data.gemfire.cache.server.Neptune.port=41414
```

Then, we can simplify the `@ClientCacheApplication` class to...

Spring `@ClientCacheApplication` class

```
@SpringBootApplication
@ClientCacheApplication
@EnablePools(pools = {
    @EnablePool(name = "VenusPool"),
    @EnablePool(name = "SaturnPool"),
    @EnablePool(name = "NeptunePool")
})
class ClientApplication { .. }
```

And, the `@CacheServerApplication` class as...

Spring `@CacheServerApplication` class

```
@SpringBootApplication
@CacheServerApplication(name = "SpringApplication")
@EnableCacheServers(servers = {
    @EnableCacheServer(name = "Venus"),
    @EnableCacheServer(name = "Saturn"),
    @EnableCacheServer(name = "Neptune")
})
class ServerApplication { .. }
```

The example above illustrates why it is import to "name" your Annotation-based beans (other than it is required in certain cases). Doing so makes it possible to reference the bean in a Spring context from XML, properties and even Java. It is even possible to inject Annotation-defined beans into an application class, for whatever purpose; for example...

```
@Component
class MyApplicationComponent {

  @Resource(name = "Saturn")
  CacheServer saturnCacheServer;

  ...
}
```

Likewise, naming a Annotated-defined bean allows you to code a `Configurer` to customize a specific, "named" bean since the `beanName` is 1 of 2 arguments passed to the callback.

Often times, an associated Annotation attribute property takes 2 forms: a "named" property along with an "unnamed" property.

For example...

```
spring.data.gemfire.cache.server.bind-address=10.105.20.1
spring.data.gemfire.cache.server.Venus.bind-address=10.105.20.2
spring.data.gemfire.cache.server.Saturn...
spring.data.gemfire.cache.server.Neptune...
```

While there are 3 named `CacheServers` above, there is 1 unnamed `CacheServer` property that serves as the default value for any unspecified value for that property even for "named" `CacheServers`. So, while "Venus" sets and overrides its own `bind-address`, "Saturn" and "Neptune" inherit from the unnamed `spring.data.gemfire.cache.server.bind-address` property.

Refer to an Annotation's *Javadoc* for which Annotation attributes support property-based configuration, and whether they support "named" properties over just "default", unnamed properties.

6.5.1. `Properties` of `Properties`

Of course, in Spring fashion, you can even express `Properties` in terms of other `Properties`, whether that is using a *Spring Boot* `application.properties` file or by using the `@Value` annotation in your Java class...

Properties of Properties

```
spring.data.gemfire.cache.server.port=${geode.cache.server.port:40404}
```

Or, in Java...

```
  @Bean
  CacheServerConfigurer cacheServerPortConfigurer(
          @Value("${geode.cache.server.port:${some.other.property:40404}}") int
cacheServerPort) {

      ...
  }
}
```

Property placeholder nesting can be arbitrarily deep.

6.6. Configuring embedded services

Apache Geode provides the ability to start many different embedded services required by an application depending on the use case.

6.6.1. Configuring an embedded Locator

As mentioned previously, Apache Geode Locators are used by clients to connect with and find servers in a cluster as well as by new members joining an existing cluster to find other peers.

It is often convenient for application developers as they are developing their *Spring Boot*, *Spring Data Geode* applications to startup up a small cluster of 2 or 3 Apache Geode servers. Rather than starting a separate Locator process, a user can simply annotate her `@CacheServerApplication` class with `@EnableLocator`.

Spring, Apache Geode `CacheServer` application running an embedded Locator

```
@SpringBootApplication
@CacheServerApplication
@EnableLocator
class ServerApplication { .. }
```

The `@EnableLocator` annotation starts and embedded Locator in the Spring, Apache Geode `CacheServer` application process running on "*localhost*", listening on the default Locator port **10334**. It is possible to customize the `host` (a.k.a bind address) and `port` that the embedded Locator binds to using the corresponding Annotation attributes.

Then, it is possible to start other *Spring Boot*, `@CacheServerApplication` enabled applications connecting to this Locator with...

Spring, Apache Geode `CacheServer` application connecting to a Locator

```
@SpringBootApplication
@CacheServerApplication(locators = "localhost[10334]")
class ServerApplication { .. }
```

You may even combine both application classes shown above into a single class and use your IDE feature to create different run profile configurations to create and run different instances of the same class with slightly modified configuration using Java System Properties...

Spring `CacheServer` application running an embedded Locator and connecting to the Locator

```
@SpringBootApplication
@CacheServerApplication(locators = "localhost[10334]")
public class ServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServerApplication.class);
  }

  @EnableLocator
  @Profile("embedded-locator")
  static class Configuration {
  }
}
```

Then, for each run profile, a user simply sets and changes the following System properties...

IDE run profile configuration

```
spring.data.gemfire.cache.name=SpringCacheServerOne
spring.data.gemfire.cache.server.port=41414
spring.profiles.active=embedded-locator
```

Only 1 of the run profiles for the `ServerApplication` class should be set with the `-Dspring.profiles.active=embedded-locator` Java System Property. Then, simply change the `name` and `cache.server.port` for each of the other run profiles and you'll have yourself a small cluster/distributed system of Geode Servers.

NOTE	The `@EnableLocator` annotation was meant to be a development-time annotation only and not something an application developer should use in production. It is recommended that Locators be stand-alone, independent processes in the cluster.

More details on how Apache Geode Locators work can be found here.

6.6.2. Configuring an embedded Manager

An Apache Geode Manager is another peer member/node in the cluster that is responsible for "management" activities. Management activities include things like creating Regions, Indexes, DiskStores, etc. The Manager allows a JMX-enabled client (e.g. *Gfsh* shell tool) to connect to the manager to manage the cluster. It is also possible to connect to a Manager with JDK provided tools like *JConsole* or *JVisualVM*, given these are both JMX-enabled clients as well.

Perhaps we would also like to make our Spring `@CacheServerApplication` shown above a Manager as well. Simply annotate your Spring `@Configurtion` or `@SpringBootApplication` class with `@EnableManager` and you are done. By default, the Manager binds to "*localhost*" listening on the default Apache Geode Manager port **1099**. Several aspects of the Manager can be configured with the Annotation attributes or corresponding properties.

Spring `CacheServer` application running an embedded Manager

```
@SpringBootApplication
@CacheServerApplication(locators = "localhost[10334]")
public class ServerApplication {

  public static void main(String[] args) {
    SpringApplication.run(ServerApplication.class);
  }

  @EnableLocator
  @EnableManager
  @Profile("embedded-locator-manager")
  static class Configuration {
  }
}
```

With the above class, you can even use *Gfsh* to connect to this server and manage it.

```
$ gfsh
    _________________________     __
   / _____/ ______/ ______/ /____/ /
  / /  __/ /___  /_____  / _____  /
 / /__/ / ____/  _____/ / /    / /
/______/_/      /______/_/    /_/    1.2.1

Monitor and Manage Apache Geode

gfsh>connect
Connecting to Locator at [host=localhost, port=10334] ..
Connecting to Manager at [host=10.99.199.5, port=1099] ..
Successfully connected to: [host=10.99.199.5, port=1099]

gfsh>list members
         Name          | Id
---------------------- | ---------------------------------------------------
SpringCacheServerOne   | 10.99.199.5(SpringCacheServerOne:14842)<ec><v0>:1024
SpringCacheServerTwo   | 10.99.199.5(SpringCacheServerTwo:14844)<v1>:1025
SpringCacheServerThree | 10.99.199.5(SpringCacheServerThree:14846)<v2>:1026
```

Because we also have the embedded Locator enabled, we were able to connect indirectly to the Manager through the Locator. The Locator allows JMX clients to connect and find a Manager node in the cluster. If none exist, the Locator will assume the role of the Manager. However, if no Locator existed, then we would need to connect directly to the Manager using...

Gfsh `connect` command connecting directly to the Manager

```
gfsh>connect --jmx-manager=localhost[1099]
```

NOTE Like the `@EnableLocator` annotation, the `@EnableManager` annotation was also meant to be a development-time only and not something an application developer should use in production. It is recommended that Managers, like Locators, be stand-alone, independent processes in the cluster.

More details on Apache Geode Management and Monitoring can be found here.

6.6.3. Configuring the embedded HTTP Server

Apache Geode is also capable of running an embedded HTTP server. The current implementation is backed by Eclipse Jetty.

The embedded HTTP server is used to host Apache Geode's Management (Admin) REST API (not a publicly advertised API), the Developer REST API and the Pulse Monitoring Web Application.

However, to use any of these Apache Geode provided Web applications, you must have a full installation of Apache Geode installed on your system, and you must set the `GEMFIRE` environment variable to your installation directory.

To enable the embedded HTTP server, simply add the `@EnableHttpService` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class...

Spring `CacheServer` application running an embedded HTTP server

```
@SpringBootApplication
@CacheServerApplication
@EnableHttpService
public class ServerApplication { .. }
```

By default, the embedded HTTP server listens on port **7070** for HTTP client requests. Of course, you can use the Annotation attributes or corresponding configuration properties to adjust the configuration as needed.

Follow the links above for more details on HTTP support.

6.6.4. Configuring the embedded Memcached Server (Gemcached)

Apache Geode also implements the Memcached protocol with the ability to service Memcached clients. That is Memcached clients can connect to an Apache Geode cluster and perform Memcached operations as if the Apache Geode Servers in the cluster were actual Memcached Servers.

To enable the embedded Memcached Service, simply add the `@EnableMemcachedServer` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class...

Spring `CacheServer` application running an embedded Memcached Server

```
@SpringBootApplication
@CacheServerApplication
@EnabledMemcachedServer
public class ServerApplication { .. }
```

More details on Apache Geode's *Gemcached* service can be found here.

6.6.5. Configuring the embedded Redis Server

Apache Geode also implements the Redis Server protocol, which enables Redis clients to communicate with a cluster of Apache Geode Servers and issue Redis commands. As of this writing, the Redis Server protocol support in Apache Geode is still experimental.

To enable the embedded Redis Service, simply add the `@EnableRedisServer` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class...

Spring `CacheServer` application running an embedded Redis Server

```
@SpringBootApplication
@CacheServerApplication
@EnableRedisServer
public class ServerApplication { .. }
```

More details on Apache Geode's Redis Adapter can be found here.

6.7. Configuring Logging

Often times it is necessary to turn up logging in order to understand exactly what Apache Geode is doing and when.

To enable *Logging*, simply annotate your application class with `@EnableLogging` and set the appropriate attributes or associated properties...

Spring `ClientCache` application with Logging enabled

```
@SpringBootApplication
@ClientCacheApplication
@EnableLogging(logLevel="info", logFile=
"/absolute/file/system/path/to/application.log)
public class ClientApplication { .. }
```

While the `logLevel` attribute can be specified with all the cache-based application annotations (e.g. `@ClientCacheApplication(logLevel="info")`), it is easier to customize logging behavior with the `@EnableLogging` annotation.

See the `@EnableLogging` annotation *Javadoc* for more details.

6.8. Configuring Statistics

To gain even deeper insight into Apache Geode during runtime, an application developer can enable *Statistics*. Gathering statistical data facilitates system analysis and troubleshooting when complex problems occur, which are often distributed in nature and timing is a factor.

When *Statistics* are enabled, a user can use Apache Geode's VSD (*Visual Statistics Display*) tool to analyze the statistical data that is collected.

To enable *Statistics*, simply annotate your application class with `@EnableStatistics`...

Spring `ClientCache` application with Statistics enabled

```
@SpringBootApplication
@ClientCacheApplication
@EnableStatistics
public class ClientApplication { .. }
```

Enabling *Statistics* on a server is particularly valuable when evaluating performance, which is as simple as annotating your `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableStatistics`.

Use the `@EnableStatistics` annotation attributes or associated properties to customize the *Statistics* gathering and collection process.

See the `@EnableStatistics` annotation *Javadoc* for more details.

More details on Apache Geode's *Statistics* can be found here.

6.9. Configuring PDX

One of the more powerful features of Apache Geode is PDX Serialization. While a complete discussion on PDX is well beyond the scope of this document, serialization using PDX is a much better alternative to *Java Serialization*, with the following benefits...

1. PDX uses a centralized *Type Registry* to keep the serialized bytes of an object more compact.
2. PDX is a neutral serialization format allowing both Java and Native Clients to operate on the same data set.
3. PDX supports versioning and allows object fields to be added or removed with affecting applications using either older or newer versions of the PDX serialized, application domain object types that change, without data loss.
4. PDX allows object fields to be accessed individually or in OQL query projections and predicates without the object needing to be de-serialized first.

In general, serialization in Apache Geode is needed anytime data is transferred to/from clients and servers or between peers in a cluster for normal distribution and replication processes as well as when data is overflowed or persisted to disk.

To enable PDX, simply annotate your application class with `@EnablePdx`...

Spring `ClientCache` application with PDX enabled

```
@SpringBootApplication
@ClientCacheApplication
@EnablePdx
public class ClientApplication { .. }
```

Typically, an application's domain object types will either implement the `org.apache.geode.pdx.PdxSerializable` interface, or an application developer will choose to implement and register the non-invasive `org.apache.geode.pdx.PdxSerializer` interface to handle the application domain object types that need to be serialized.

Unfortunately, Apache Geode only allows one `PdxSerializer` to be registered, which suggests that all application domain object types should be handled by a "single" `PdxSerializer` instance. But, that is a serious anti-pattern and foolish practice to be sure.

Even though only a single `PdxSerializer` instance can be registered with Apache Geode, it makes sense to create a `PdxSerializer` implementation per application domain object type.

By using the Composite Software Design Pattern, the application developer can provide an implementation of the `PdxSerializer` interface that aggregates of all application domain object type-specific `PdxSerializer` instances but acts as a single `PdxSerializer` instance, and register it.

You can declare this *Composite* `PdxSerializer` as a managed bean in the Spring context and refer to this *Composite* `PdxSerializer` by bean name in the `@EnablePdx` annotation using the `serializerBeanName` attribute. *Spring Data Geode* will take care of registering it with Apache Geode on the user's behalf.

Spring `ClientCache` *application with PDX enabled, using a custom, composite* `PdxSerializer`

```
@SpringBootApplication
@ClientCacheApplication
@EnablePdx(serializerBeanName = "compositePdxSerializer")
public class ClientApplication {

  @Bean
  PdxSerializer compositePdxSerializer() {
      return new CompositePdxSerializerBuilder()...
  }
}
```

It is also possible to declare Apache Geode's `org.apache.geode.pdx.ReflectionBasedAutoSerializer` as a bean definition in a Spring context. Alternatively, you can use *Spring Data Geode's* more robust, `org.springframework.data.gemfire.mapping.MappingPdxSerializer`, which uses *Spring Data* mapping meta-data and infrastructure applied to the serialization process for more efficient handling than reflection alone.

Many other aspects and features of PDX can be adjusted with the `@EnablePdx` annotation attributes or associated configuration properties.

See the `@EnablePdx` annotation *Javadoc* for more details.

6.10. Configuring SSL

Equally important to serializing data to be transferred over-the-wire is securing the data while in transit. Of course, the common way to accomplish this in *Java* is using the *Secure Sockets Extension* (SSE) and *Transport Layer Security* (TLS).

To enable SSL, simply annotate your application class with `@EnableSsl` and set the necessary SSL configuration attributes (e.g. keystores, usernames/passwords, etc)...

Spring `ClientCache` application with SSL enabled

```
@SpringBootApplication
@ClientCacheApplication
@EnableSsl
public class ClientApplication { .. }
```

Different Apache Geode components: `GATEWAY`, `HTTP`, `JMX`, `LOCATOR`, `SERVER` can be individually configured with SSL, or they can all be collectively configured all at once to use SSL using the `CLUSTER` enumerated value.

It is easy to specify which Apache Geode components that the SSL configuration settings should applied to using the nested `@EnableSsl` annotation `Component` enum…

Spring `ClientCache` application with SSL enabled by Aache Geode component

```
@SpringBootApplication
@ClientCacheApplication
@EnableSsl(components = { GATEWAY, LOCATOR, SERVER })
public class ClientApplication { .. }
```

In addition component-level SSL configuration, `ciphers`, `protocols` and `keystore`/`truststore` information can also be specified using the corresponding Annotation attribute or associated configuration properties.

See the `@EnableSsl` annotation *Javadoc* for more details.

More details on Apache Geode SSL support can be found here.

6.11. Configuring GemFire Properties

While many of the gemfire.properties are conveniently encapsulated in and abstracted with an Annotation in the SDG Annotation-based configuration model, the less commonly used *Geode Properties* are still accessible from the `@EnableGemFireProperties` annotation.

Using the `@EnableGemFireProperties` annotation on your application class is convenient and a nice alternative to creating a `gemfire.properties` file or setting *Geode Properties* as Java System properties on the command-line when launching your application.

TIP It is recommended that these *Geode Properties* be set in a `gemfire.properties` file when deploying your application to production. But, at development-time, it can be convenient to set these properties individually, as needed, for prototyping and/or testing purposes.

A few examples of some of the less common *Geode Properties* that a user usually need not worry about include, but are not limited to: `ack-wait-threshold`, `disable-tcp`, `socket-buffer-size`, etc.

To individually set any *Geode Property*, simply annotate your application class with

`@EnableGemFireProperties` and set the *Geode Properties* you want to change from the default, out-of-the-box value set by Apache Geode…

Spring `ClientCache` application with specific Geode Properties set

```
@SpringBootApplication
@ClientCacheApplication
@EnableGemFireProperties(conflateEvents = true, socketBufferSize = 16384)
public class ClientApplication { .. }
```

Keep in mind, some of the *Geode Properties* are client specific (e.g. `conflateEvents`) while others are server specific (e.g. `distributedSystemId`, `enableNetworkPartitionDetection`, `enforceUniqueHost`, `memberTimeout`, `redundancyZone`).

More details on Apache Geode properties can be found here.

6.12. Configuring Regions

So far, outside of PDX, our discussion has centered around configuring Apache Geode's more administrative functions: creating a cache instance, starting embedded services, enabling Logging, Statistics and SSL, using `gemfire.properties` to affect very low-level configuration and behavior. While all these configuration options are important, none of them relate directly to the application. In other words, we still need some place to store our application data and make it generally available and accessible.

Apache Geode organizes data in a cache into Regions. You can think of a Region as a table in a relational database. Generally, a Region should only store a single type of object making it more conducive for building effective `Indexes`. We will talk about Indexing later.

Previously, *Spring Data Geode* users needed to explicitly define and declare the Regions used in their applications to store data by writing very verbose Spring configuration meta-data, whether a user was using SDG's `FactoryBeans` from the API in Spring's Java-based container configuration…

Example Region bean definition using Spring Java-based container configuration

```
@Configuration
class GeodeConfiguration {

  @Bean("Example")
  PartitionedRegionFactoryBean exampleRegion(GemFireCache gemfireCache) {

      PartitionedRegionFactoryBean<Long, Example> exampleRegion =
          new PartitionedRegionFactoryBean<>();

      exampleRegion.setCache(gemfireCache);
      exampleRegion.setClose(false);
      exampleRegion.setPersistent(true);

      return exampleRegion;
  }

  ...
}
```

Or, using XML...

Example Region bean definition using the SDG XML Namespace

```
  <gfe:partitioned-region id="exampleRegion" name="Example" persistent="true">
     ...
  </gfe:partitioned-region>
```

While neither Java nor XML configuration is hard to do, it is cumbersome, especially if an application has a large number of Regions that need to be defined. Many relational database-based applications can literally have 100s or even 1000s of tables.

Ugh!

Now users can define and configure Regions based on their application domain objects (entities). No longer does a user need to explicitly define `Region` bean definitions in Spring configuration meta-data, unless finer-grained control is required.

To simplify Region creation, *Spring Data Geode* combines the use of SD *Repositories* with the expressive power of Annotation-based configuration using the **new** `@EnableEntityDefinedRegions` annotation.

NOTE	Most *Spring Data* application developers should already be familiar with the *Spring Data Repository* abstraction and *Spring Data Geode's* implementation/extension of the SD *Repository abstraction*, which has been specifically customized to optimize data access operations for Apache Geode.

First, an application developer starts by defining the application domain objects...

Application domain object type modeling a Book

```
@Region("Books")
class Book {

  @Id
  private ISBN isbn;

  private Author author;

  private Category category;

  private LocalDate releaseDate;

  private Publisher publisher;

  private String title;

}
```

Next, the application developer would define a basic *Repository* for `Books` by extending *Spring Data Commons* `org.springframework.data.repository.CrudRepository` interface...

Repository for Books

```
interface BookRepository extends CrudRepository<Book, ISBN> { .. }
```

The `org.springframe.data.repository.CrudRepository` is a Data Access Object (DAO) providing basic data access operations (CRUD) along with support for simple queries (e.g. `findById(..)`). The user can define additional, more sophisticated queries simply by declaring query methods on the *Repository* interface (e.g. `List<BooK> findByAuthor(Author author);`).

Under-the-hood, *Spring Data Geode* provides an implementation of this interface when the Spring container is bootstrapped. SDG will even implement the query methods defined by the user so long as the user follows simple conventions.

Now, when a user defined the `Book` class, she also specified the Region in which instances of `Book` will be mapped and stored by declaring the *Spring Data Geode* mapping annotation, `@Region` on the entity's type. Of course, if the entity type (i.e. `Book`) referenced in the type parameter of the *Repository* interface (i.e. `BookRepository`) is not annotated with `@Region`, the name is derived from the simple class name of the entity type (i.e. "Book").

Spring Data Geode uses the mapping context containing mapping meta-data for all the entities defined in your application to determine all the Regions that will be needed at runtime.

To enable and use this feature, simply annotate the application class with `@EnableEntityDefinedRegions`...

Entity-defined Region Configuration

```
@SpringBootApplication
@ClientCacheApplication
@EnableGemfireRepositories
@EnableEntityDefinedRegions(basePackages = "example.app.domain")
class ClientApplication { .. }
```

TIP	Creating Regions from entity classes is the most useful when using *Spring Data Repositories* in your application. *Spring Data Geode's Repository* support is enabled with the `@EnableGemfireRepositories` annotation.

By default, the `@EnableEntityDefinedRegions` annotation will scan for entity classes recursively starting from the package of the configuration class on which the `@EnableEntityDefinedRegions` annotation is defined.

However, it is common to limit the search during the scan by setting the `basePackages` attribute with the package names containing your application entity classes.

Alternatively, a user can use the more type-safe `basePackageClasses` attribute for specifying the package to scan by setting the attribute to an entity type in the package containing the entity's class, or by using a non-entity placeholder class in the package specifically created for identifying the package to scan. For example...

Entity-defined Region Configuration using the Entity class type

```
@SpringBootApplication
@ClientCacheApplication
@EnableGemfireRepositories
@EnableEntityDefinedRegions(basePackageClasses = {
    example.app.books.domain.Book.class,
    example.app.customers.domain.Customer.class
})
class ClientApplication { .. }
```

In addition to specifying the location where to begin the scan, like Spring's `@ComponentScan` annotation, a user can specify include and exclude filters with all the same semantics of the `org.springframework.context.annotation.ComponentScan.Filter` annotation.

See the `@EnableEntityDefinedRegion` annotation *Javadoc* for more details.

6.12.1. Configuring Type-specific Regions

Apache Geode supports many different types of Regions. Each type corresponds to the Region's `DataPolicy`, which determines exactly how the data in the Region will be managed (e.g. distributed/replicated, etc).

NOTE	Other configuration settings also can affect how data is managed like the Region's `scope`. See Storage and Distribution Options in the Apache Geode User Guide for more details.

When user annotate their application domain object types with the generic `@Region` mapping annotation, *Spring Data Geode* will decide which type of `Region` to create. SDG's default strategy takes the cache type into consideration when determining the type of `Region` to create.

For example, if the application was declared as a `ClientCache` using the `@ClientCacheApplication` annotation, then SDG would create a client `PROXY Region`. Or, if the application was declared as a peer `Cache` using either the `@PeerCacheApplication` or `@CacheServerApplication` annotations, then SDG would create a server `PARTITION Region`.

Of course, an application developer is always able to override the default when necessary. To override the default applied by *Spring Data Geode*, 4 new Region mapping annotations have been introduced:

- `ClientRegion`
- `LocalRegion`
- `PartitionRegion`
- `ReplicateRegion`

The `ClientRegion` mapping annotation is specific to client applications. All other Region mapping annotations listed above can only be used in server applications.

It is sometimes necessary for client applications to create and use "local-only" Regions, perhaps to aggregate data from other Regions in order to analyze the data locally and carry out some function performed by the application for the user. In this case, the data may not necessarily need to be distributed back to the server, not unless other applications need access to the results. This Region might even be temporary and discarded after use, which could be accomplished with Idle-Timeout (TTI) and Time-To-Live (TTL) expiration policies on the Region itself (NOTE: this is independent of and different from "entry" TTI/TTL expiration policies).

In any case, if a user wanted to create a local-only, client Region where the data is not gong to be distributed to a corresponding Region with the same name on the server, the user would specify the `@ClientRegion` mapping annotation and set the `shortcut` attribute to `ClientRegionShortcut.LOCAL`...

Spring `ClientCache` application with a local-only, client Region

```
@ClientRegion(shortcut = ClientRegionShortcut.LOCAL)
class ClientLocalEntityType { .. }
```

All `Region` type-specific annotations provide additional attributes that are both common across `Region` types as well as specific to only that type of `Region` (e.g. the `collocatedWith` and `redundantCopies` attributes in the `PartitionRegion` annotation apply to `PARTITION` Regions only).

More details on Apache Geode Region Types can be found here.

6.12.2. Configuring Eviction

Managing data with Apache Geode is an active task. More than likely, tuning will be required and a combination of features (e.g. both Eviction and Expiration) will need to be employed to effectively manage your data in memory with Apache Geode.

Given that Apache Geode is an *In-Memory Data Grid* (IMDG), data is managed in "memory" and distributed to other nodes that participate in a cluster in order to minimize latency, maximize throughput and ensure that data is highly available. Since not all of an application's data is going to typically fit in memory, even across an entire cluster of nodes, much less on a single node, capacity can be increased by adding new nodes to the cluster. This is commonly referred to as linear scale-out (rather than scaling up, which means to add more memory, more CPU, more disk, more network bandwidth, basically more of every system resource in order to handle the load).

Still, even with a cluster of nodes, it is usually imperative that only the most important data be kept in memory. Running out-of-memory, or even nearing full capacity, is rarely, if ever, a good thing. Stop-the-world GCs or worse, `OutOfMemoryErrors`, will bring your application to a screaming halt.

So, to help manage memory and keep the most important data around, Apache Geode supports LRU-based *Eviction*. That is, Apache Geode evicts Region entries based on when those entries were last accessed by using the *Least Recently Used* algorithm.

To enable *Eviction*, simply annotate the application class with `@EnableEviction`...

Spring application with Eviction enabled

```
@SpringBootApplication
@PeerCacheApplication
@EnableEviction(policies = {
    @EvictionPolicy(regionNames = "Books", action = EvictionActionType.INVALIDATE),
    @EvictionPolicy(regionNames = { "Customers", "Orders" }, maximum = 90,
        action = EvictionActionType.OVERFLOW_TO_DISK,
        type = EvictonPolicyType.HEAP_PERCENTAGE)
})
class ServerApplication { .. }
```

Eviction policies are usually set on the Regions in the server(s).

As shown above, the `policies` attribute can specify 1 or more nested `@EvictionPolicy` annotations, each 1 individually catered to 1 or more Regions where the Eviction policy needs to be applied.

Additionally, a user can reference a custom implementation of Apache Geode's `org.apache.geode.cache.util.ObjectSizer` interface defined as a bean in the Spring context and referenced by name using the `objectSizerName` attribute.

An `ObjectSizer` allows the user to define the criteria used to evaluate and determine the the size of objects stored in the Region.

See the `@EnableEviction` annotation *Javadoc* for a complete list of Eviction configuration options.

More details on Apache Geode Eviction can be found here.

6.12.3. Configuring Expiration

Along with *Eviction*, *Expiration* can also be used to manage memory by allowing entries stored in a Region to expire. Both *Time-to-Live* (TTL) and *Idle-Timeout* (TTI) based entry expiration policies are supported in Apache Geode.

Spring Data Geode's Annotation-based Expiration configuration is based on earlier, existing entry expiration annotation support that was added in *Spring Data Geode* many releases ago.

Essentially, *Spring Data Geode's* Expiration annotation support is based on a provided, custom implementation of Apache Geode's `org.apache.geode.cache.CustomExpiry` interface. This custom implementation inspects the user's application domain objects stored in a Region for the presence of type-level Expiration annotations.

Spring Data Geode provides the following Expiration annotations used on application domain object types, out-of-the-box...

- `Expiration`
- `IdleTimeoutExpiration`
- `TimeToLiveExpiration`

An application domain object type can be annotated with 1 or more of the Expiration annotations, like so...

Applicaton domain object specific Expiration policy

```
@Region("Books")
@TimeToLiveExpiration(timeout = 30000, action = "INVALIDATE")
class Book { .. }
```

To enable *Expiration*, simply annotate the application class with `@EnableExpiration`...

Spring application with Expiration enabled

```
@SpringBootApplication
@PeerCacheApplication
@EnableExpiration
class ServerApplication { .. }
```

In addition to application domain object type-level Expiration policies, individual Expiration policies on a Region-by-Region basis can be configured directly with the `@EnableExpiration` annotation as well.

Spring application with global Expiration policies

```
@SpringBootApplication
@PeerCacheApplication
@EnableExpiration(policies = {
    @ExpirationPolicy(regionNames = "Books", types = ExpirationType.TIME_TO_LIVE),
    @ExpirationPolicy(regionNames = { "Customers", "Orders" }, timeout = 30000,
        action = ExpirationActionType.LOCAL_DESTROY)
})
class ServerApplication { .. }
```

Expiration policies are usually set on the Regions in the server(s).

See the `@EnableExpiration` annotation *Javadoc* for a complete list of Expiration configuration options.

More details on Apache Geode Expiration can be found here.

6.12.4. Configuring Compression

In addition to *Eviction* and *Expiration* a user may also configure his or her data Regions to use Compression in order to reduce memory consumption.

Apache Geode allows users to compress in-memory Region values using pluggable `Compressors`, or different compression codecs. Out-of-the-box, Apache Geode uses Google's Snappy library.

To enable Compression support, simply annotate the application class with `@EnableCompression`...

Spring application with Compression enabled

```
@SpringBootApplication
@ClientCacheApplication
@EnableCompression(compressorBeanName = "MyCompressor", regionNames = { "Customers",
"Orders" })
class ClientApplication { .. }
```

NOTE Neither the `compressorBeanName` nor the `regionNames` attributes are required.

The `compressorBeanName` defaults to "SnappyCompressor" enabling Apache Geode's provided `SnappyCompressor` by default.

The `regionNames` attribute is an array of Region names specifying the Regions that will have compression enabled. By default, all Regions will compress values if the `regionNames` attribute is not explicitly set.

TIP Alternatively, a user may use the `spring.data.gemfire.cache.compression.compressor-bean-name` and `spring.data.gemfire.cache.compression.region-names` properties in the `application.properties` file to set and configure the values of these `@EnableCompression` annotation attributes.

WARNING	To use Apache Geode's Region Compression feature, you must include the `org.iq80.snappy:snappy` dependency in your *Maven* `pom.xml` or `build.gradle` file when using *Gradle*. This is only necessary if you use Apache Geode's default, out-of-the-box support for Region Compression, which uses the `SnappyCompressor` by default. Of course, if you are using another compression library, you will need to include dependencies for that compression library on your application's classpath. Additionally, you will need to implement Apache Geode's `Compressors` to adapt your compression library of choice, define it as a bean in the *Spring* context, and then set the `compressorBeanName` to this custom bean definition.

See the `@EnableCompression` annotation *Javadoc* for more details.

More details on Apache Geode Compression can be found here.

6.12.5. Configuring Off-Heap

Another effective means for reducing pressure on the JVM's Heap memory and minimize GC activity is to use Apache Geode's *Off-Heap* memory support. Rather than storing Region entries on the JVM Heap, entries are stored in the system's main memory.

To enable *Off-Heap* support, simple annotate the application class with `@EnableOffHeap`...

Spring application with Off-Heap enabled

```
@SpringBootApplication
@PeerCacheApplication
@EnableOffHeap(memorySize = 8192m regionNames = { "Customers", "Orders" })
class ServerApplication { .. }
```

The `memorySize` attribute is required. The value for the `memorySize` attribute specifies the amount of main memory a Region is allowed to use in either megabytes (`m`) or gigabytes (`g`).

The `regionNames` attribute is an array of Region names specifying the Regions that will store entries in main memory. By default, all Regions will use main memory if the `regionNames` attribute is not explicitly set.

TIP	Alternatively, a user may use the `spring.data.gemfire.cache.off-heap.memory-size` and `spring.data.gemfire.cache.off-heap.region-names` properties in the `application.properties` file to set and configure the values of these `@EnableOffHeap` annotation attributes.

See the `@EnableOffHeap` annotation *Javadoc* for more details.

6.12.6. Configuring Indexes

There is not much use in storing data in Regions unless the data can be retrieved.

In addition to `Region.get(key)` operations, particularly when the key of the value of interest is

known in advance, data is commonly retrieved by executing queries on the Regions containing the data. With Apache Geode, queries are written using the *Object Query Language* (OQL), and the specific data set that a client wishes to access is expressed in the query's predicate (e.g. `SELECT * FROM /Books b WHERE b.author.name = 'Jon Doe'`).

Generally, querying without Indexes is not very efficient. When executing queries without an Index, Apache Geode performs the equivalent of a full table scan.

Indexes are created and maintained for fields on objects used in query predicates to match the data of interests, expressed by the query's projection. Different types of Indexes can be created, such as Key and Hash Indexes.

Spring Data Geode makes it very easy to create Indexes on Regions where the data is stored and accessed. Rather than explicitly declaring `Index` bean definitions using Spring config as before...

Index bean definition using Java config

```
@Bean("BookIsbnIndex")
IndexFactoryBean bookIsbnIndex(GemFireCache gemfireCache) {

    IndexFactoryBean bookIsbnIndex = new IndexFactoryBean();

    bookIsbnIndex.setCache(gemfireCache);
    bookIsbnIndex.setName("BookIsbnIndex");
    bookIsbnIndex.setExpression("isbn");
    bookIsbnIndex.setFrom("/Books"));
    bookIsbnIndex.setType(IndexType.KEY);

    return bookIsbnIndex;
}
```

Or, in XML...

Index bean definition using XML

```
  <gfe:index id="BooksIsbnIndex" expression="isbn" from="/Books" type="KEY"/>
```

Indexes can now be created directly from the fields defined on application domain object types that a user knows will be used in query predicates to speedup those queries. Indexes will be applied even for OQL queries generated from user-defined query methods on an application's *Repository* interfaces.

Re-using the example `Book` class from above, we can annotate the fields on `Book` that we know will be used in queries we define with query methods in the `BookRepository` interface...

Application domain object type modeling a Book using Indexes

```
@Region("Books")
class Book {

  @Id
  private ISBN isbn;

  @Indexed
  private Author author;

  private Category category;

  private LocalDate releaseDate;

  private Publisher publisher;

  @LuceneIndexed
  private String title;

}
```

In our new `Book` class definition, we annotated the `author` field with `@Indexed` and the `title` field with `@LuceneIndexed`. Also, the `isbn` field had previously been annotated with *Spring Data's* `@Id` annotation, which identifies the field containing the unique identifier for `Book` instances, and in *Spring Data Geode*, the `@Id` annotated field or property is used as the key in the Region when storing the entry.

- `@Id` annotated fields/properties result in the creation of an Apache Geode KEY Index.
- `@Indexed` annotated fields/properties result in the creation of an Apache Geode HASH Index (default).
- `@LuceneIndexed` annotated fields/properties result in the creation of an Apache Geode Lucene Index, used in text-based searches with Apache Geode's Lucene Integration and support.

When the `@Indexed` annotation is used without setting any attributes, the Index `name`, `expression`, and `fromClause` are derived from the field/property of the object on which the `@Indexed` annotation has been added. The `expression` is exactly the name of the field or property. The `fromClause` is derived from the `@Region` annotation on the object's class (or the simple name of the domain object class if the `@Region` annotation was not specified).

Of course, any of the `@Indexed` annotation attributes may be explicitly set to override the default values provided by *Spring Data Geode*.

Application domain object type modeling a Book using cutomized Indexes

```
@Region("Books")
class Book {

  @Id
  private ISBN isbn;

  @Indexed(name = "BookAuthorNameIndex", expression = "author.name", type =
"FUNCTIONAL")
  private Author author;

  private Category category;

  private LocalDate releaseDate;

  private Publisher publisher;

  @LuceneIndexed(name = "BookTitleIndex", destory = true)
  private String title;

}
```

The `name` of the Index, which is auto-generated when not explicitly set, is also used as the name of the bean registered in the Spring context for the Index. If necessary, this Index bean could even be injected by name into another application component.

The generated name of the Index follows the pattern: `<Region Name><Field/Property Name><Index Type>Idx`. For example, the name of the `author` Index would be, “BooksAuthorHashIdx”.

To enable Indexing, simply annotate the application class with `@EnableIndexing`…

Spring application with Indexing enabled

```
@SpringBootApplication
@PeerCacheApplication
@EnableEntityDefinedRegions
@EnableIndexing
class ServerApplication { .. }
```

NOTE

The `@EnablingIndexing` annotation has no effect unless the `@EnableEntityDefinedRegions` is also declared. Essentially, Indexes are defined from entity class types, and entity classes must be scanned in order to inspect the entity’s fields and properties for the presence of Index annotations. Without this scan, Index annotations would not be found. It is also imperative that you limit the scope of the scan.

While Lucene queries are not supported on *Spring Data Geode Repositories* (yet), SDG does provide comprehensive support for Apache Geode Lucene queries using the familiar Spring *Template*

pattern.

Finally, we close with a few extra things to keep in mind when using Indexes:

1. While OQL Indexes are not required to execute OQL Queries, Lucene Indexes are required to execute Lucene, text-based searches.
2. In addition, OQL Indexes are not persisted to disk; they are maintained only in memory. So, when an Apache Geode node is restarted, the Index must be rebuilt.
3. You also need to be aware of the overhead associated in maintaining Indexes, particularly since an Index is stored exclusively in memory, and especially when Region entries are updated. Index "maintenance" can be configured as an asynchronous task.

Another optimization that can be utilized when re-starting your Spring application where Indexes have to be rebuilt is to first define all the Indexes upfront and then create them all at once, which, in *Spring Data Geode*, happens when the Spring context is refreshed.

Indexes can be defined upfront then created all at once by setting the `define` attribute on the `@EnableIndexing` annotation to `true`.

See Creating Multiple Indexes at Once in Apache Geode's User Guide for more details.

Creating sensible Indexes is an important task since it is possible for an Index to do more harm than good if not properly designed.

See both the `@Indexed` annotation and `@LuceneIndexed` annotation *Javadoc* for complete list of configuration options.

More details on Apache Geode OQL Queries can be found here.

More details on Apache Geode Indexes can be found here.

More details on Apache Geode Lucene Queries can be found here.

6.12.7. Configuring Disk Stores

Regions can be configured to persist data to disk. Regions can also be configured to overflow data to disk when Region entries are evicted. In both cases, a `DiskStore` is required to persist or overflow the data. When an explicit `DiskStore` has not been set on a Region with persistence or overflow configured, then Apache Geode will use the "DEFAULT" `DiskStore`.

However, it is possible and recommended to define Region-specific `DiskStores` when persisting or overflowing data to disk.

Spring Data Geode provides Annotation support for defining and creating application Region `DiskStores` by annotating the application class with the `@EnableDiskStore` and `@EnableDiskStores` annotations.

TIP	`@EnableDiskStores` is a composite annotation for aggregating 1 or more `@EnableDiskStore` annotations.

For example, while `Book` product information might mostly consist of reference data, from some external data source (e.g. Amazon), `Order` data is most likely going to be transactional in nature and something the application is going to need to retain, maybe even overflow to disk if the transaction volume is high enough, or so any Book publisher hopes, anyway.

Using the `@EnableDiskStore` annotation, I can define and create a `DiskStore` as follows...

Spring application defining a `DiskStore`

```
@SpringBootApplication
@PeerCacheApplication
@EnableDiskStore(name = "OrdersDiskStore", autoCompact = true, compactionThreshold =
70,
    maxOplogSize = 512, diskDirectories = @DiskDiretory(location =
"/absolute/path/to/order/disk/files"))
class ServerApplication { .. }
```

Again, more than 1 `DiskStore` can be defined using the composite, `@EnableDiskStores` annotation.

Like other Annotations in *Spring Data Geode's* Annotation-based configuration model, both `@EnableDiskStore` and `@EnableDiskStores` have many attributes along with associated configuration properties to apply additional configuration to `DiskStores` created at runtime.

Additionally, the `@EnableDiskStores` annotation defines certain common `DiskStore` attributes that apply to all `DiskStores` created from `@EnableDiskStore` annotations composed with the `@EnableDiskStores` annotation itself. Individual `DiskStore` configuration can override a particular global setting, but the `@EnableDiskStores` annotation defines common configuration attributes for all `DiskStores` out of convenience.

Spring Data Geode also provides the `DiskStoreConfigurer` callback interface that can be declared in Java config and used instead of configuration properties to customize a `DiskStore` at runtime...

Spring application with custom DiskStore configuration

```
@SpringBootApplication
@PeerCacheApplication
@EnableDiskStore(name = "OrdersDiskStore", autoCompact = true, compactionThreshold =
70,
    maxOplogSize = 512, diskDirectories = @DiskDiretory(location =
"/absolute/path/to/order/disk/files"))
class ServerApplication {

  @Bean
  DiskStoreConfigurer ordersDiskStoreDiretoryConfigurer(
          @Value("${orders.disk.store.location}") String location) {

      return (beanName, diskStoreFactoryBean) -> {

          if ("OrdersDiskStore".equals(beanName) {
              diskStoreFactoryBean.setDiskDirs(Collections.singletonList(new DiskDir
(location));
          }
      }
  }
}
```

See the `@EnableDiskStore` and `@EnableDiskStores` annotation *Javadoc* for more details on the available attributes as well as associated configuration properties.

More details on Apache Geode Region Persistence and Overflow (using Disk Stores) can be found here.

6.13. Configuring Continuous Queries

Another very important and useful feature of Apache Geode is Continuous Querying.

In a world of Internet-enabled things, events and streams of data are coming in from everywhere. Being able to handle and process a large stream of data and react to events in real-time, as they happen, is becoming an increasingly important requirement for many applications. One example is self-driving vehicles. Being able to receive, filter, transform, analyze and act on data in real-time is a key differentiator and characteristic of real-time enabled applications.

Fortunately, Apache Geode was ahead of its time in this regard. Using *Continuous Queries* (CQ) a client application can express the data, or events it is interested in and register listeners to handle and process the events as they arrive. The data that a client application may be interested in is expressed in a OQL query, where the query predicate is used to filter, or identify the data of interests. When data is changed or added and it matches the criteria defined in the query predicate of the registered CQ, the client application is notified.

Spring Data Geode makes defining and registering CQs along with an associated listener to handler and process CQ events without all the cruft of Apache Geode's plumbing, a non-event (no pun

intended). SDG's new Annotation-based configuration for CQs builds on the already existing *Continuous Query* support in the Continuous Query Listener Container.

For instance, say a Book publisher wants to register interests in and receive notification anytime orders (demand) for a `Book` exceeds the current inventory (supply), then the publisher's print application might register the following CQ...

Spring `ClientCache` application with registered CQ and Listener.

```
@SpringBootApplication
@ClientCacheApplication(subcriptionEnabled = true)
@EnableContinuousQueries
class PublisherPrintApplication {

    @ContinuousQuery(name = "DemandExceedsSupply", query =
       "SELECT book.* FROM /Books book, /Inventory inventory
        WHERE book.title = 'How to crush it in the Book business like Amazon"
        AND inventory.isbn = book.isbn
        AND inventory.available < (
            SELECT sum(order.lineItems.quantity)
            FROM /Orders order
            WHERE order.status = 'pending'
            AND order.lineItems.isbn = book.isbn
        )
    ")
    void handleSupplyProblem(CqEvent event) {
        // start printing more Books, fast!
    }
}
```

To enable *Continuous Queries*, simply annotate your application class with `@EnableContinuousQueries`.

Defining *Continuous Queries* is as simple as annotating any Spring `@Component` annotated POJO class methods with the `@ContinuousQuery` annotation, in similar fashion to SDG's Function annotated POJO methods. A method defined with a CQ using the `@ContinuousQuery` annotation will be called anytime data matching the query predicate is added or changed.

Additionally, the POJO method signature should adhere to the requirements outlined in the section on ContinuousQueryListener and ContinuousQueryListenerAdapter.

See the `@EnableContinuousQueries` and `@ContinuousQuery` annotation *Javadoc* for more details on available attributes and configuration settings.

More details on *Spring Data Geode's* Continuous Query support can be found here.

More details on Apache Geode's Continuous Queries can be found here.

6.14. Configuring Spring's Cache Abstraction

With *Spring Data Geode*, Apache Geode can be used as a caching provider in Spring's Cache Abstraction.

In *Spring's Cache Abstraction*, the caching annotations (e.g. `@Cacheable`) identify the cache on which a cache lookup is performed before invoking a potentially expensive operation, or where the results of an application service method are cached after the operation is invoked.

In *Spring Data Geode*, a Spring `Cache` corresponds directly to a Region. The Region must exist before any `@Cacheable` application service method is called. This is true for any of Spring's caching annotations (i.e. `@Cacheable`, `@CachePut` and `@CacheEvict`) that identify the cache to use in the operation.

For instance, our publisher's Point-of-Sale (POS) application might have a feature to determine, or lookup the `Price` of a `Book` during a sales transaction.

```
@Service
class PointOfSaleService

  @Cacheable("BookPrices")
  Price runPriceCheckFor(Book book) {
      ...
  }

  @Transactional
  Receipt checkout(Order order) {
      ...
  }

  ...
}
```

To make the application developer's life easier when using *Spring Data Geode* and Apache Geode with *Spring's Cache Abstraction*, 2 new features have been added to the **new** Annotation-based configuration model.

Given the following Spring caching configuration...

Enabling Caching using Apache Geode with Spring Data Geode

```
@EnableCaching
class CachingConfiguration {

  @Bean
  GemfireCacheManager cacheManager(GemFireCache gemfireCache) {

      GemfireCacheManager cacheManager = new GemfireCacheManager();

      cacheManager.setCache(gemfireCache);

      return cacheManager;
  }

  @Bean("BookPricesCache")
  PartitionedRegionFactoryBean<Book, Price> bookPricesRegion(GemFireCache
gemfireCache) {

    PartitionedRegionFactoryBean<Book, Price> bookPricesRegion =
        new PartitionedRegionFactoryBean<>();

    bookPricesRegion.setCache(gemfireCache);
    bookPricesRegion.setClose(false);
    bookPricesRegion.setPersistent(false);

    return bookPricesRegion;
  }

  @Bean("PointOfSaleService")
  PointOfSaleService pointOfSaleService(..) {
      return new PointOfSaleService(..);
  }
}
```

Using *Spring Data Geode's* new features, the same caching configuration can be simplified to...

Enabling GemFire Caching

```
@EnableGemfireCaching
@EnableCachingDefinedRegions
class CachingConfiguration {

  @Bean("PointOfSaleService")
  PointOfSaleService pointOfSaleService(..) {
      return new PointOfSaleService(..);
  }
}
```

First, the `@EnableGemfireCaching` annotation replaces both the Spring `EnableCaching` annotation along

with the need to declare an explicit `cacheManager` bean definition in the Spring config.

Second, the `@EnableCachingDefinedRegions` annotation, like the `@EnableEntityDefinedRegions` annotation described in Configuring Regions, inspects all the Spring caching annotated application service components to identify all the caches that will be needed by the application at runtime and creates Regions in Apache Geode for these caches on application startup.

The Region created is local to the application process that created the Region. If the application is a peer `Cache`, then the Region will only exist on the application node. If the application is a `ClientCache`, then SDG creates a client `PROXY` Region and expects that a Region with the same name already exists on the servers in the cluster.

NOTE	SDG cannot determine the cache required by a service method using a Spring `CacheResolver` to resolve the cache used in the operation at runtime.

NOTE	SDG does not currently identify *JCache*, JSR-107 cache annotations used on application service components. Refer to the core *Spring Framework Reference Guide* for the equivalent Spring caching annotation to use in place of *JCache*, JSR-107 caching annotations.

Refer to the section, Support for the Spring Cache Abstraction for more details on using Apache Geode as a caching provider in *Spring's Cache Abstraction*.

More details on *Spring's Cache Abstraction* can be found here.

6.15. Configuring Cluster Configuration Push

This may be the most exciting **new** feature in *Spring Data Geode*.

When a client application class is annotated with `@EnableClusterConfiguration`, any Regions or Indexes defined and declared as beans in the Spring context by the client application are "pushed" to the cluster of servers to which the client is connected. Not only that, but this "push" is performed in such a way that Apache Geode will remember the configuration pushed by the client. If all the nodes in the cluster go down, they will come back up with the same configuration as before.

In a sense, this feature is not much different than if a user were to use *Gfsh* to create the Regions and Indexes on all the servers in the cluster. Except now, with *Spring Data Geode*, users does **not** need to use *Gfsh* to create Regions and Indexes. The user's *Spring Boot* application, enabled with the power of *Spring Data Geode*, already contains all the configuration meta-data SDG needs to create Regions and Indexes for the user.

When users are using the *Spring Data Repository* abstraction, we know all the Regions (e.g. `@Region` annotated entity types) and Indexes (e.g. `@Indexed` annotated entity fields and properties) that the users' application will need. When users are using *Spring's Cache Abstraction*, we also know all the Regions for all the caches identified in the caching annotations that the application is going to need. Essentially, the user is already telling us everything we need to know just by developing her application with the entire *Spring Framework* and all of its provided services, infrastructure, etc, whether expressed in Annotation meta-data, Java, XML or otherwise, and whether for

configuration, for mapping, or whatever purpose.

The user can focus on her application business logic along with using the framework provided services and supporting infrastructure (e.g. *Spring Data Repositories*, *Spring's Transaction Management*, *Spring Caching*, etc) and *Spring Data Geode* will take care of all the Apache Geode plumbing required by those framework services on the user's behalf.

Pushing configuration from the client to the servers in the cluster and having the cluster remember it is made possible in part by the use of Apache Geode's *Cluster Configuration* service. Apache Geode's *Cluster Configuration* service is also the same service used by *Gfsh* to record schema-related changes issued by the user to the cluster from the shell.

Of course, since the cluster "remembers" the prior configuration pushed by a client from a previous run perhaps, *Spring Data Geode* is careful not to stomp on any existing Regions and Indexes already defined in the servers. This is especially important when Regions already contain data.

NOTE	Currently there is no option to overwrite any existing Region or Index definitions. To recreate a Region or Index, the user must use *Gfsh* to destroy the Region or Index first and then restart the client application so that configuration will be pushed up to the server again. Alternatively a user can just use *Gfsh* to (re-)define the Regions and Indexes manually.

NOTE	Unlike *Gfsh*, *Spring Data Geode* only supports the creation of Regions and Indexes on the servers from a client. For advanced configuration and use cases, *Gfsh* should be used to manage the cluster.

For a moment, imagine the power expressed in the following configuration…

Spring `ClientCache` *application*

```
@SpringBootApplication
@ClientCacheApplication
@EnableCachingDefinedRegions
@EnableEntityDefinedRegions
@EnableIndexing
@EnableGemfireCaching
@EnableGemfireRepositories
@EnableClusterConfiguration
class ClientApplication { .. }
```

An application developer instantly gets a *Spring Boot*, Apache Geode `ClientCache` application using *Spring Data Repositories* with *Spring's Cache Abstraction*, using Apache Geode as the caching provider, where Regions and Indexes are not only created on the client, but pushed to the servers in the cluster.

All the application developer need do is define the application's domain model objects annotated with mapping and Index annotations, define Repository interfaces supporting basic data access operations and querying for each of the entity types, define the service components containing the business logic manipulating the entities, declare the appropriate annotations on service methods

that require caching, transactional behavior, etc, and the developer is in business. Nothing the user did in this case pertains to infrastructure and plumbing required in the application's back-end services (e.g. Apache Geode). Database users have similar features, no Spring, Apache Geode developers can too.

When combined with a couple more *Spring Data Geode* Annotations...

- `@EnableContinuousQueries`
- `@EnableGemfireFunctionExecutions`
- `@EnableGemfireCacheTransactions`

Then, the application is really going to start to take flight.

See the `@EnableClusterConfiguration` annotation *Javadoc* for more details.

6.16. Configuring Security

Without a doubt, application *Security* is extremely important and *Spring Data Geode* provides comprehensive support for securing both Apache Geode clients and servers.

Recently, Apache Geode introduced a new Integrated Security framework, replacing its old Authentication and Authorization Security model, for handling authentication and authorization. One of the main features and benefits of this new Security framework is that it integrates with Apache Shiro and can therefore delegate both authentication and authorization requests to Apache Shiro when enforcing security.

The following demonstrates how *Spring Data Geode* can simplify Apache Geode's Security story even further.

6.16.1. Configuring Server Security

There are several different ways in which a user can configure Security for servers in an Apache Geode cluster.

1. Implement the Apache Geode `org.apache.geode.security.SecurityManager` interface and set Apache Geode's `security-manager` property to refer to your application `SecurityManager` implementation by the FQCN. Alternatively, users can construct and initialize an instance of their `SecurityManager` implementation and set it with CacheFactory.setSecurityManager(:SecurityManager) method when creating an instance of an Apache Geode peer `Cache`.
2. Create an Apache Shiro `shiro.ini` file with the *users*, *roles* and *permissions* defined for your application, then set the Apache Geode `security-shiro-init` property to refer to this `shiro.ini` file, which must be available in the `CLASSPATH`.
3. Using just Apache Shiro, annotate your *Spring Boot* application class with *Spring Data Geode's* **new** `@EnableSecurity` annotation and define 1 or more Apache Shiro `Realms` (as needed) as beans in the Spring context for accessing your application's Security meta-data (i.e. *authorized users*, *roles* and *permissions*), and your done!

The problem with the first approach is that a user must implement his/her own `SecurityManager`,

which can be quite tedious and error prone. Implementing a custom `SecurityManager` does afford a user some flexibility in accessing Security meta-data from whatever data source stores the meta-data, such as LDAP or even a proprietary, internal data source, but then that is a problem already solved by configuring and using Apache Shiro `Realms`, which is more universally known and non-Apache Geode specific.

TIP See Apache Geode's Security examples for Authentication and Authorization as 1 possible way to implement your own custom, application specific `SecurityManager`.

The second approach using an Apache Shiro INI file is marginally better, but a user still needs to be familiar with the INI file format in the first place. Additionally, an INI file is static and not easily updatable at runtime.

The third approach is the most ideal since it adheres to widely known and industry accepted concepts (i.e. Apache Shiro's Security framework) and is easy to setup…

Spring server application using Apache Shiro

```
@SpringBootApplication
@CacheServerApplication
@EnableSecurity
class ServerApplication {

  @Bean
  PropertiesRealm shiroRealm() {
      PropertiesRealm propertiesRealm = new PropertiesRealm();
      propertiesRealm.setResourcePath("classpath:shiro.properties");
      propertiesRealm.setPermissionResolver(new GeodePermissionResolver());
      return propertiesRealm;
  }
}
```

NOTE The configured `Realm` shown in the example above could have easily been any of Apache Shiro's supported `Realms` out-of-the-box (ActiveDirectory, JDBC, JNDI, LDAP, or even a `Realm` supporting the INI format) or even a custom implementation of an Apache Shiro `Realm` implemented by the user. See Apache Shiro's documentation on Realms for more details.

When Apache Shiro is on the `CLASSPATH` of the servers in the cluster and 1 or more Apache Shiro `Realms` have been defined as beans in the Spring context, *Spring Data Geode* will detect this configuration and use Apache Shiro as the Security provider to secure your Apache Geode servers when the `@EnableSecurity` annotation is used.

TIP Earlier, information was posted on *Spring Data Geode's* support for Apache Geode's **new** Integrated Security framework using Apache Shiro in this spring.io blob post.

See the `@EnableSecurity` annotation *Javadoc* for more details on available attributes and associated configuration properties.

More details on Apache Geode Security can be found here.

6.16.2. Configuring Client Security

The Security story would not be complete without discussing how to secure Spring-based, Apache Geode cache client applications.

Apache Geode's process to securing a client application is, well, rather involved. In a nutshell, a user essentially needs to...

1. Provide an implementation of the `org.apache.geode.security.AuthInitialize` interface.
2. Set the Apache Geode `security-client-auth-init` (System) property to refer to the custom, application-provided `AuthInitialize` interface.
3. And finally, a user would typically specify the user credentials in a proprietary, Apache Geode `gfsecurity.properties` file.

Spring Data Geode simplifies all of that using the same `@EnableSecurity` annotation as applied to server applications. In other words, the same `@EnableSecurity` annotation handles Security for both client and server applications. This makes it easier for users when they decide to switch their applications from an embedded peer `Cache` application to a `ClientCache` application, for instance. Simply change the SDG annotation from `@PeerCacheApplication` or `@CacheServerApplication` to `@ClientCacheApplication` and you are done.

Effectively, all a user need do on the client is...

Spring client application using `@EnableSecurity`

```
@SpringBootApplication
@ClientCacheApplication
@EnableSecurity
class ClientApplication { .. }
```

Then define the familiar *Spring Boot* `application.properties` file containing the required *username* and *password* Security properties and you are all set.

Spring Boot `application.properties` *file with the required Security credentials*

```
spring.data.gemfire.security.username=jackBlack
spring.data.gemfire.security.password=b@cK!nB1@cK
```

That was easy!

TIP By default, *Spring Boot* can find an `application.properties` file when placed in the root of the application's `CLASSPATH`. Of course, Spring supports may ways to to locate resources using its Resource abstraction.

See the `@EnableSecurity` annotation *Javadoc* for more details on available attributes and associated configuration properties.

More details on Apache Geode Security can be found here.

6.17. Configuration Tips

The following tips will help users get the most out of using the **new** Annotation-based configuration model.

6.18. Configuration Organization

As we saw in the section on *Configuring Cluster Configuration Push*, when many Apache Geode and/or *Spring Data Geode* features are enabled using Annotations, we start to stack a lot of Annotations on the Spring `@Configuration` or `@SpringBootApplication` class. In this situation, it makes sense to start compartmentalizing the configuration a bit.

For instance, given...

Spring `ClientCache` application with the kitchen sink to boot

```
@SpringBootApplication
@ClientCacheApplication
@EnableContinuousQueries
@EnableCachingDefinedRegions
@EnableEntityDefinedRegions
@EnableIndexing
@EnableGemfireCaching
@EnableGemfireFunctionExecutions
@EnableGemfireRepositories
@EnableGemfireCacheTransactions
@EnableClusterConfiguration
class ClientApplication { .. }
```

We could break this configuration down by concern. For example...

Spring `ClientCache` application with the kitchen sink to boot

```
@SpringBootApplication
@Import({ CachingConfiguration.class, GeodeConfiguration.class,
    QueriesAndFunctionsConfiguration.class, RepositoryConfiguration.class })
class ClientApplication { .. }

@EnableGemfireCaching
@EnableCachingDefinedRegions
class CachingConfiguration { .. }

@ClientCacheApplication
@EnableClusterConfiguration
@EnableGemfireCacheTransactions
class GeodeConfiguration { .. }

@EnableContinuousQueries
@EnableGemfireFunctionExecutions
class QueriesAndFunctionsConfiguration {

   @ContinuousQuery(..)
   void processCqEvent(CqEvent event) {
       ...
   }
}

@EnableGemfireRepositories
@EnableEntityDefinedRegions
@EnableIndexing
class RepositoryConfiguration { .. }
```

Spring does not care. Organize your application configuration as you see fit.

6.19. Additional Configuration-based Annotations

SDG Annotations you never heard of...

The following SDG Annotations were not discussed in this reference documentation either because the Annotation supports a deprecated feature of Apache Geode, or there are better, alternative ways to accomplishing the function that the Annotation provides.

- `@EnableAuth` - enable Apache Geode's old Authentication/Authorization Security model. (*Deprecated*; use Apache Geode's new *Integrated Security* framework discussed here).
- `@EnableAutoRegionLookup` - Not recommended. Essentially, this Annotation supports finding Regions defined in external configuration meta-data (e.g. `cache.xml`, or *Cluster Configuration* when applied to a server) and registers those Regions as beans in the Spring context automatically. Users should generally prefer Spring config when using Spring and *Spring Data Geode*. See Configuring Regions and Configuring Cluster Configuration Push instead.

- `@EnableBeanFactoryLocator` - enables the SDG `GemfireBeanFactoryLocator` feature, which is only useful, again, when using external configuration meta-data (e.g. `cache.xml`). For example, if a user defines a `CacheLoader` on a Region defined in `cache.xml`, the user can still auto-wire this `CacheLoader` with say, a relational database `DataSource` bean defined in Spring confif. This Annotation takes advantage of this SDG feature and might be useful for users who have a large amount of legacy configuration meta-data, like `cache.xml` files.
- `@EnableGemFireAsLastResource` - is actually discussed in Global - JTA Transaction Management with Apache Geode.
- `@EnableMcast` - enables Apache Geode's old peer discovery mechanism using UDP-based Multi-cast Networking. (*Deprecated*; users should be using Apache Geode Locators instead; see Configuring Locators.
- `@EnableRegionDataAccessTracing` - is useful for debugging purposes; the Annotation enables tracing for all data access operations performed on a Region by registering an AOP Aspect that proxies all Regions declared as beans in the Spring context, intercepting the Region op and logging the event.

6.20. Conclusion

As we learned in the previous sections, there is a tremendous amount of power provided by *Spring Data Geode*'s **new** Annotation-based configuration model. Hopefully, it lives up to its goal of making it easier for users to get started quickly when using Apache Geode with Spring.

Keep in mind when using the new Annotations that it does not preclude you, the application developer, from using Java config, or even XML, if you prefer. You can even combine all 3 approaches by using Spring's `@Import` and `@ImportResource` annotations on a Spring `@Configuration` or `@SpringBootApplication` class, if you like. The moment you explicitly provide a bean definition that would otherwise be provided by *Spring Data Geode* using an Annotation, the Annotation-based configuration backs away.

In certain cases you may even need to fallback to Java config, as in the `Configurers` case, to handle more complex or conditional configuration logic that is not easily expressed in or cannot be accomplished using Annotations. Do not be alarmed; this is to be expected.

For example, another case where Java config or XML will be needed is when configuring Apache Geode WAN components, which currently do not have any Annotation configuration support. However, defining and registering WAN components is as simple as using the `org.springframework.data.gemfire.wan.GatewaReceiverFactoryBean` and `org.springframework.data.gemfire.wan.GatewaySenderFactoryBean` API classes in Java configuration on your Spring `@Configuration` or `@SpringBootApplication` classes (recommended).

The Annotations were not meant to handle every situation; the Annotations were meant to help application developers **get up and running** as **quickly** and as **easily** as possible, especially during development.

Chapter 7. Working with Apache Geode APIs

Once the Apache Geode 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 DAO exception hierarchy. This chapter also covers support for dependency injection of Geode managed objects.

7.1. GemfireTemplate

As with many other high-level abstractions provided by *Spring* projects, *Spring Data Geode* provides a **template** to simplify Geode data access. The class provides several **one-liner** methods containing common Region operations, but also has the ability to **execute** code against the native Geode API without having to deal with Geode checked exceptions by using a `GemfireCallback`.

The template class requires a Geode `Region` instance, and once configured, is thread-safe and can be reused across multiple application classes:

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

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

```
template.execute(new GemfireCallback<Iterable<String>>() {
    public Iterable<String> doInGemfire(Region region) throws GemFireCheckedException,
GemFireException {
        Region<String, String> localRegion = (Region<String, String>) region;

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

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

For accessing the full power of the Apache Geode query language, a developer can use the `find` and `findUnique` methods, 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.

7.2. 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 the exception handling case, the *Spring Framework* provides a technology agnostic and consistent exception hierarchy that abstracts the application from proprietary, and usually "checked", exceptions to a set of focused runtime exceptions.

As mentioned in *Spring Framework's* documentation, Exception translation can be applied transparently to your Data Access Objects (DAO) through the use of the `@Repository` annotation and AOP by defining a `PersistenceExceptionTranslationPostProcessor` bean. The same exception translation functionality is enabled when using Geode as long as the `CacheFactoryBean` is declared, e.g. using either a `<gfe:cache/>` or `<gfe:client-cache>` declaration, which acts as an exception translator and is automatically detected by the *Spring* infrastructure and used accordingly.

7.3. 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 Apache Geode, *Spring Data Geode* provides a dedicated, per-cache, `PlatformTransactionManager` that, once declared, allows Region operations to be executed atomically through *Spring*:

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

NOTE

The example above can be simplified even further by eliminating the `cache-ref` attribute if the Geode cache is defined under the default name, `gemfireCache`. As with the other *Spring Data Geode* 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, Apache Geode 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.

7.4. Global, JTA Transaction Management

It is also possible for Apache Geode 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), Apache Geode is **not** an XA compliant resource. Therefore, Apache Geode 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 Apache Geode 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 Apache Geode 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. Apache Geode) in such an arrangement.

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

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

2) Referring to Step 5 in Geode's documentation, *Spring Data Geode'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 Geode's documentation and let *Spring Data Geode* work its magic. All you need do is annotate your *Spring* `@Configuration` class with *Spring Data Geode's* **new** `@EnableGemFireAsLastResource` annotation and a combination of *Spring's* Transaction Management infrastructure and *Spring Data Geode'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 Geode'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 Geode's* `@EnableGemFireAsLastResource` annotation imports configuration containing 2 Aspect bean definitions that handles the Geode `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 + Geode API yourself, as shown in the Geode 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 Geode'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 Apache Geode in JTA transactions, see here.

For more details on configuring Apache Geode as a "*Last Resource*", see here.

7.5. Continuous Query (CQ)

A powerful functionality offered by Apache Geode is Continuous Query (or CQ). In short, CQ allows one to create and register an OQL query, and then automatically be notified when new data that

gets added to Geode matches the query predicate. *Spring Data Geode* 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 the *Spring Framework*; in fact, users familiar with the JMS support in *Spring*, should feel right at home.

Basically *Spring Data Geode* allows methods on POJOs to become end-points for CQ. Simply define the query and indicate the method that should be called to be notified when there is a match. *Spring Data Geode* takes care of the rest. This is very similar to Java EE's message-driven bean style, but without any requirement for base class or interface implementations, based on Apache Geode.

NOTE	Currently, Continuous Query is only supported in Geode's client/server topology. Additionally, the client Pool used is required to have the subscription enabled. Please refer to the Geode documentation for more information.

7.5.1. Continuous Query Listener Container

Spring Data Geode simplifies creation, registration, life-cycle and dispatch of CQ events by taking care of the infrastructure around CQ with the use of SDG's `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 provided in the *Spring Framework* with its Message-driven POJOs (MDPs).

The SDG `ContinuousQueryListenerContainer` acts as an event (or message) listener container; it is used to receive the events from the registered CQs and invoke 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 delegate the boilerplate Geode infrastructure concerns to the framework.

The listener container is fully customizable. A developer can chose either to use the CQ thread to perform the dispatch (synchronous delivery) or a new thread (from an existing pool) 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, the developer 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 proper `TaskExecutor` to take advantage of its runtime.

7.5.2. The `ContinuousQueryListener` and `ContinuousQueryListenerAdapter`

The `ContinuousQueryListenerAdapter` class is the final component in *Spring Data Geode* CQ support. In a nutshell, class allows you to expose almost **any** implementing class as an EDP with minimal constraints. `ContinuousQueryListenerAdapter` implements the `ContinuousQueryListener` interface, a simple listener interface similar to Geode's 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 throwable);
     void handleQuery(CqQuery cq);
     void handleEvent(CqEvent event, Operation baseOp, byte[] deltaValue);
     void handleEvent(CqEvent event, Operation baseOp, Operation queryOp, Object key,
Object newValue);
}
```

```
package example;

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

In particular, note how the above implementation of the `EventDelegate` interface has **no** Geode dependencies at all. It truly is a POJO that we can and will make into an EDP via the following configuration.

NOTE	the class does not have to implement an interface; an interface is only used to better showcase the decoupling between the contract and the implementation.

```
<?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:client-cache/>

    <gfe:pool 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 /SomeRegion"/>
       <gfe:listener ref="another-listener" query="SELECT * FROM /AnotherRegion" name
="myQuery" method="handleQuery"/>
    </gfe:cq-listener-container>

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

NOTE

The example above shows a few 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 Geode* 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 CQ listeners -->
      <set>
        <bean class=
"org.springframework.data.gemfire.listener.ContinuousQueryDefinition" >
              <constructor-arg value="SELECT * FROM /SomeRegion" />
              <constructor-arg ref="eventListener"/>
        </bean>
      </set>
    </property>
</bean>
```

Each time an event is received, the adapter automatically performs type translation between the Geode 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).

7.6. Wiring `Declarable` Components

Apache Geode XML configuration (usually referred to as `cache.xml`) allows **user** objects to be declared as part of the configuration. Usually these objects are `CacheLoaders` or other pluggable callback components supported by Geode. Using native Geode 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 when 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 collaborators.

However, if you are starting a green field project, it is recommended that you configure Cache, Region, and other pluggable Geode 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

A developer can configure custom types entirely through *Spring* as mentioned in Configuring a Region. That way, a developer 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 Geode* offers a base class (`WiringDeclarableSupport`) that allows Geode user objects to be wired through a **template** bean definition or, in case that is missing, perform auto-wiring through the *Spring* IoC 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 Geode 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 Apache Geode.

7.6.1. Configuration using template bean definitions

When used, `WiringDeclarableSupport` tries to first locate an existing bean definition and use that as the 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:

```
class DBLoader extends WiringDeclarableSupport implements CacheLoader {

  private DataSource dataSource;

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       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
">

  <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 Geode. For cases where the bean name uses a different convention, one can pass in the `bean-name` parameter in the Geode 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: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
">

  <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).

7.6.2. Configuration using auto-wiring and annotations

By default, if no bean definition is found, `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, a developer can also use JDK 5 annotations to provide additional information to the auto-wiring process.

TIP 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:

```
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 since the class is auto-wired -->
</cache-loader>
```

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

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

</beans>
```

By using the JSR-330 annotations, the `CacheLoader` 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 easily be configured and changed through the *Spring* container without touching the `DBLoader` code.

7.7. Support for the Spring Cache Abstraction

Spring Data Geode provides an implementation of the *Spring* Cache Abstraction to position Apache Geode as a *caching provider* in Spring's caching infrastructure.

To use Apache Geode as a backing implementation, a "*caching provider*" *in Spring's Cache Abstraction*, 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">

  <!-- enable declarative caching -->
  <cache:annotation-driven/>

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

  <!-- declare GemfireCacheManager; must have a bean ID of 'cacheManager' -->
  <bean id="cacheManager" class=
"org.springframework.data.gemfire.cache.GemfireCacheManager"
      p:cache-ref="gemfire-cache">

</beans>
```

NOTE	The `cache-ref` attribute on the `CacheManager` bean definition is not necessary if the default cache bean name is used (i.e. "gemfireCache"), i.e. `<gfe:cache>` without an explicit ID.

When the `GemfireCacheManager` (Singleton) bean instance is declared and declarative caching is enabled (either in XML with `<cache:annotation-driven/>` or in JavaConfig with *Spring's* `@EnableCaching` annotation), the *Spring* caching annotations (e.g. `@Cacheable`) identify the "caches" that will cache data in-memory using Geode Regions.

These caches (i.e. Regions) must exist before the caching annotations that use them otherwise an error will occur.

By way of example, suppose you have a Customer Service application with a `CustomerService` application component that performs caching...

```
@Service
class CustomerService {

@Cacheable(cacheNames="Accounts", key="#customer.id")
Account createAccount(Customer customer) {
  ...
}
```

Then you will need the following config.

XML:

```
<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">

  <!-- enable declarative caching -->
  <cache:annotation-driven/>

  <bean id="cacheManager" class=
"org.springframework.data.gemfire.cache.GemfireCacheManager">

  <gfe:cache/>

  <gfe:partitioned-region id="accountsRegion" name="Accounts" persistent="true" ...>
    ...
  </gfe:partitioned-region>
</beans>
```

JavaConfig:

```
@Configuration
@EnableCaching
class ApplicationConfiguration {

  @Bean
  CacheFactoryBean gemfireCache() {
    return new CacheFactoryBean();
  }

  @Bean
  GemfireCacheManager cacheManager() {
    GemfireCacheManager cacheManager = GemfireCacheManager();
    cacheManager.setCache(gemfireCache());
    return cacheManager;
  }

  @Bean("Accounts")
  PartitionedRegionFactoryBean accountsRegion() {
    PartitionedRegionFactoryBean accounts = new PartitionedRegionFactoryBean();

    accounts.setCache(gemfireCache());
    accounts.setClose(false);
    accounts.setPersistent(true);

    return accounts;
  }
}
```

Of course, you are free to choose whatever Region type you like (e.g. REPLICATE, PARTITION, LOCAL, etc).

For more details on *Spring's Cache Abstraction*, again, please refer to the documentation.

Chapter 8. Working with Apache Geode Serialization

To improve overall performance of the Apache Geode In-memory Data Grid, Geode supports a dedicated serialization protocol, called PDX, that is both faster and offers more compact results over standard Java serialization in addition to works transparently across various language platforms (Java, C++, .NET). Please refer to PDX Serialization Features and PDX Serialization Internals for more details.

This chapter discusses the various ways in which *Spring Data Geode* simplifies and improves Geode's custom serialization in Java.

8.1. Wiring deserialized instances

It is fairly common for serialized objects to have transient data. Transient data is often dependent on the system or environment where it lives at a certain point in time. For instance, a `DataSource` is environment specific. Serializing such information is useless, and potentially even dangerous, since it is local to a certain VM/machine. For such cases, *Spring Data Geode* offers a special `Instantiator` that performs wiring for each new instance created by Geode 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 falling back to autowiring otherwise.

Please refer to the previous section (Wiring `Declarable` Components) for more details on wiring functionality.

To use this SDG `Instantiator`, simply declare it as a 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 *Spring* container startup, once it is being initialized, the `Instantiator` will, by default, register itself with the Geode serialization system and perform wiring on all instances of `SomeDataSerializableClass` created by Geode during deserialization.

8.2. Auto-generating custom `Instantiators`

For data intensive applications, a large number of instances might be created on each machine as data flows in. Out-of-the-box, Geode 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 Geode* allows the automatic generation of `Instatiator` classes which instantiate a new type (using the default constructor) without the use of reflection:

```
<bean id="instantiatorFactory" 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 generates two `Instantiators` for two classes, namely `CustomTypeA` and `CustomTypeB` and registers them with Geode, under user id `1025` and `1026`. The two `Instantiators` avoid the use of reflection and create the instances directly through Java code.

Chapter 9. POJO mapping

9.1. Entity Mapping

Spring Data Geode provides support to map entities that will be stored in a Region in the Geode In-Memory Data Grid. The mapping metadata is defined using annotations on application domain classes just like this:

Example 1. Mapping a domain class to a Geode Region

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

  @Id Long id;

  String firstname;
  String lastname;

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

  ···
}
```

The first thing you notice here is the `@Region` annotation that can be used to customize the Region in which an instance of the `Person` class is stored. The `@Id` annotation can be used to annotate the property that shall be used as the cache (Region) key, identifying the Region entry. The `@PersistenceConstructor` annotation helps to disambiguate multiple, potentially available constructors taking parameters and explicitly marking the constructor annotated as the constructor to be used to construct entities. In an application domain class with no or only a single constructor you can omit the annotation.

In addition to storing entities in top-level Regions, entities can be stored in Sub-Regions as well.

For instance:

```
@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 Geode Region, as defined with the *Spring Data Geode* XML namespace using the `id` or `name` attributes of the `<*-region>` element.

9.1.1. Entity Mapping by Region Type

In addition to the `@Region` annotation, *Spring Data Geode* also recognizes the Region type-specific mapping annotations: `@ClientRegion`, `@LocalRegion`, `@PartitionRegion` and `@ReplicateRegion`.

Functionally, these annotations are treated exactly the same as the generic `@Region` annotation in the SDG mapping infrastructure. However, these additional mapping annotations are useful in *Spring Data Geode's` Annotation configuration model. When combined with the* `@EnableEntityDefinedRegions` *configuration annotation on _Spring* `@Configuration` annotated class, it is possible to generate Regions in the local cache, whether the application is a client or peer.

These annotations allow you, the developer, to be more specific about what type of Region that your application entity class should be mapped to, and also has an impact on the data management policies of the Region (e.g. partition (a.k.a. sharding) vs. just replicating data).

Using these Region type-specific mapping annotations with the SDG Annotation config model saves you from having to explicitly define these Regions in config.

The details of the new Annotation configuration model will be discussed in more detail in a subsequent releaase.

9.1.2. Repository Mapping

As an 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`. See Spring Data Geode Repositories for more details.

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

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

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

Then, using each Repository individually, you can store the entity in multiple Geode Regions.

```
@Service
class CustomerService {

  CustomerRepository customerRepo;

  PersonRepository personRepo;

  Customer update(Customer customer) {
    customerRepo.save(customer);
    personRepo.save(customer);
    return customer;
  }
```

It is not difficult to imagine wrapping the `update` service method in a *Spring* managed transaction, either as a local cache transaction or a global transaction.

9.2. Mapping PDX Serializer

Spring Data Geode 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 default constructor, a singly declared constructor or an explicitly annotated constructor annotated with the `@PersistenceConstructor` annotation).

To provide values for constructor parameters it will read fields with name of the constructor parameters from the supplied PdxReader.

Example 2. Using @Value on entity constructor parameters

```
public class Person {

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

An entity class annotated in this way will have the field `foo` read from the `PdxReader` and passed to the constructor parameter value for `firstname`. The value for `lastname` will be the *Spring* bean with the name `bean`.

Chapter 10. Spring Data Geode Repositories

10.1. Introduction

Spring Data Geode provides support to use the *Spring Data Repository* abstraction to easily persist entities into Geode along with execute queries. A general introduction to the *Repository programming model* is provided here.

10.2. Spring XML Configuration

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

Example 3. Bootstrap Spring Data Geode Repositories in XML

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:gfe-data="http://www.springframework.org/schema/data/geode"
       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/geode
http://www.springframework.org/schema/data/geode/spring-data-geode.xsd>

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

</beans>
```

This configuration snippet looks for interfaces below the configured base package and creates *Repository* instances for those interfaces backed by a `SimpleGemFireRepository`.

IMPORTANT	You must have your application domain classes correctly mapped to configured Regions or the bootstrap process will fail otherwise.

10.3. Spring Java-based Configuration

Alternatively, many users prefer to use *Spring's* Java-based container configuration.

Using this approach, it is a simple matter to bootstrap *Spring Data Repositories* using the SDG `@EnableGemfireRepositories` annotation:

Example 4. Bootstrap Spring Data Geode Repositories with `@EnableGemfireRepositories`

```
@SpringBootApplication
@EnableGemfireRepositories(basePackages = "com.example.acme.repository")
class SpringApplication {
  ...
}
```

Rather than use the `basePackages` attribute, you may prefer to use the type-safe `basePackageClasses` attribute instead. The `basePackageClasses` allows you to specify the package containing all your application *Repository* classes by specifying just one of your application *Repository* interface types. Consider creating a special no-op marker class or interface in each package that serves no other purpose than to identify the location of application *Repositories* referenced by this attribute.

In addition to the `basePackage[sClasses]` attributes, like *Spring's* `@ComponentScan` annotation, the `@EnableGemfireRepositories` annotation provides *include* and *exclude* filters, based on *Spring's* `ComponentScan.Filter` type. You can use the `filterType` attribute to filter by different aspects, such as whether an application *Repository* type is annotated with a particular `Annotation` or extends a particular class type, and so on. See the `FilterType` *Javadoc* for more details.

The `@EnableGemfireRepositories` annotation also provides the ability to specify the location of named OQL queries, which reside in a Java `Properties` file, using the `namedQueriesLocation` attribute. The property name must match the name of a *Repository* query method and the property value is the OQL query you want executed when the *Repository* query method is called.

The `repositoryImplementationPostfix` attribute can be set to an alternate value (defaults to "*Impl*") if your application requires 1 or more custom *Repository* implementations. This feature is commonly used to extend the *Spring Data Repository* infrastructure in order to implement a feature not provided out-of-the-box (OOTB) by the data store (e.g. SDG).

One example of where custom *Repository* implementations are needed with Apache Geode is when performing *Joins*. *Joins* are not supported by SDG *Repositories* OOTB. With an Apache Geode `PARTITION` Region, the *Join* must be performed on collocated `PARTITION` Regions even, since Apache Geode does not support "distributed" *Joins*. In addition, the *Equi-Join* OQL Query must be performed inside a Geode Function. See here for more details on Apache Geode *Equi-Join Queries*.

Many other aspects of the SDG's *Repository* infrastructure extension maybe customized as well. See the `@EnableGemfireRepositories` *Javadoc* for more details on all configuration settings.

10.4. Executing OQL Queries

Spring Data Geode Repositories enable the definition of query methods to easily execute Geode OQL Queries against the Region the managed entity is mapped to.

Example 5. Sample Repository

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

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

  Person findByEmailAddress(String emailAddress);

  Collection<Person> findByFirstname(String firstname);

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

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

The first query method listed here will cause the following OQL query to be derived: `SELECT x FROM /People x WHERE x.emailAddress = $1`. The second query method works the same way except it's returning all entities found whereas the first query method expects a single result to be found.

In case the supported keywords are not sufficient to expresss and declare your OQL query, or the method name becomes too 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`

Keyword	Sample	Logical result
`IsFalse`, `False`	`findByActiveIsFalse()`	`x.active = false`

10.5. OQL Query Extensions using Annotations

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

One of *Spring Data Commons' Repository* infrastructure goals is to function as the lowest common denominator in order to maintain support for 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 Geode's OQL Query language extensions and preserve portability across different data stores, *Spring Data Geode* adds support for OQL Query extensions using Java Annotations. These Annotations will be ignored by other *Spring Data Repository* implementations (e.g. *Spring Data* JPA or *Spring Data Redis*) that do not have similar query language extensions.

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

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

Table 5. Supported Geode OQL extensions for Repository 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 class and corresponding Geode Region along with a `CustomerRepository` and a query method to lookup `Customers` by last name, like so...

Example 6. Sample Customers Repository

```
package ...;

import org.springframework.data.annotation.Id;
import org.springframework.data.gemfire.mapping.annotation.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 x
WHERE x.lastName = $1 LIMIT 10
```

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

As another example, suppose you have a raw `@Query` annotated query method defined in your `CustomerRepository` like so...

Example 7. 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 x WHERE x.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 can 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.

10.6. Query Post Processing

Using the Spring Data *Repository* abstraction, query method convention for defining data store specific queries (e.g. OQL) is easy and convenient. However, it is sometimes desirable to still want to inspect or even possibly modify the query "generated" from the *Repository* query method.

Since 2.0.x, *Spring Data Geode* introduces the `o.s.d.gemfire.repository.query.QueryPostProcessor` functional interface. The interface is loosely defined as follows...

Example 8. QueryPostProcessor

```
package org.springframework.data.gemfire.repository.query;

import org.springframework.core.Ordered;
import org.springframework.data.repository.Repository;
import org.springframework.data.repository.query.QueryMethod;
import ...;

@FunctionalInterface
interface QueryPostProcessor<T extends Repository, QUERY> extends Ordered {

  QUERY postProcess(QueryMethod queryMethod, QUERY query, Object... arguments);

}
```

There are additional default methods provided to allow users to compose instances of `QueryPostProcessor` very similar to how java.util.function.Function.andThen(:Function) and java.util.function.Function.compose(:Function) work.

Additionally, you will notice that the `QueryPostProcessor` interface implements the `org.springframework.core.Ordered` interface, which is useful when multiple `QueryPostProcessors` are declared and registered in the Spring context and used to create a pipeline of processing for a group of generated query method queries.

Finally, the `QueryPostProcessor` accepts type arguments corresponding to the type parameters, `T` and `QUERY`, respectively. Type of `T` extends the *Spring Data Commons* marker interface, `org.springframework.data.repository.Repository`. We will discuss this further below. All `QUERY` type parameter arguments in *Spring Data Geode's* case will be of type `java.lang.String`.

NOTE	It is useful to define the query as type `QUERY` since this `QueryPostProcessor` interface maybe ported to *Spring Data Commons* and therefore must handle all forms of queries by different data stores (e.g. JPA, MongoDB, or Redis).

As user may implement this interface to receive a callback with the query that was generated from the application `Repository` interface method when the method is called.

For example, I might want to log all queries from all application *Repository* interface definitions. I could do so using the following `QueryPostProcessor` implementation...

Example 9. LoggingQueryPostProcessor

```
package example;

import ...;

class LoggingQueryPostProcessor implements QueryPostProcessor<Repository, String>
{

  private Logger logger = Logger.getLogger("someLoggerName");

  @Override
  public String postProcess(QueryMethod queryMethod, String query, Object...
arguments) {

      String message = String.format("Executing query [%s] with arguments [%s]",
query, Arrays.toString(arguments));

      this.logger.info(message);
  }
}
```

The `LoggingQueryPostProcessor` was typed to the Spring Data `org.springframework.data.repository.Repository` marker interface, and therefore, will log all application *Repository* interface query method "generated" queries.

You could limit the scope of this logging to queries only from certain types of application *Repository* interfaces, such as, say, an `CustomerRepository`...

Example 10. CustomerRepository

```
interface CustomerRepository extends CrudRepository<Customer, Long> {

  Customer findByAccountNumber(String accountNumber);

  List<Customer> findByLastNameLike(String lastName);

}
```

Then, I could have typed the `LoggingQueryPostProcessor` specifically to the `CustomerRepository`, like so...

Example 11. CustomerLoggingQueryPostProcessor

```
class LoggingQueryPostProcessor implements QueryPostProcessor<CustomerRepository,
String> { .. }
```

As result, only queries defined in the `CustomerRepository` interface (e.g. `findByAccountNumber`) would be logged.

I might want to create a `QueryPostProcessor` for a specific query defined by a *Repository* query method. For example, say I want to "LIMIT" the OQL query generated from the `CustomerRepository.findByLastNameLike(:String)` query method to only return 5 results and I want to order the `Customers` by `firstName`, ascending. Well, then, I can define a custom `QueryPostProcessor` like so...

Example 12. OrderedLimitedCustomerByLastNameQueryPostProcessor

```
class OrderedLimitedCustomerByLastNameQueryPostProcessor implements
QueryPostProcessor<CustomerRepository, String> {

  private final int limit;

  public OrderedLimitedCustomerByLastNameQueryPostProcessor(int limit) {
    this.limit = limit;
  }

  @Override
  public String postProcess(QueryMethod queryMethod, String query, Object...
arguments) {

    return "findByLastNameLike".equals(queryMethod.getName())
      ? query.trim()
          .replace("SELECT", "SELECT DISTINCT")
          .concat(" ORDER BY firstName ASC")
          .concat(String.format(" LIMIT %d", this.limit))
      : query;
  }
}
```

While this works, it possible to achieve the same affect just using the Spring Data *Repository* convention and extensions provided by *Spring Data Geode*. For instance, the same query could be defined as...

Example 13. CustomerRepository using the convention

```
interface CustomerRepository extends CrudRepository<Customer, Long> {

  @Limit(5)
  List<Customer> findDistinctByLastNameLikeOrderByFirstNameDesc(String lastName);

}
```

However, if you do not have control over the application `CustomerRepository` interface definition, then the `QueryPostProcessor` (i.e. `OrderedLimitedCustomerByLastNameQueryPostProcessor`) is convenient.

If I want to ensure the `LoggingQueryPostProcessor` always comes after the other application-defined `QueryPostProcessors` that I may have declared and registered in the Spring `ApplicationContext`, then I can set the `order` property by overriding the `o.s.core.Ordered.getOrder()` method.

Example 14. Defining the `order` property

```
class LoggingQueryPostProcessor implements QueryPostProcessor<Repository, String>
{

  @Override
  int getOrder() {
    return 1;
  }
}

class CustomerQueryPostProcessor implements QueryPostProcessor<CustomerRepository,
String> {

  @Override
  int getOrder() {
    return 0;
  }
}
```

This ensures that I will always see the affects of the post processing applied by my other `QueryPostProcessors` before my `LoggingQueryPostProcessor` logs the query.

You can define as many `QueryPostProcessors` in the Spring `ApplicationContext` as you like and apply them in any order, to all or specific application *Repository* interfaces, and be a granular as yuo like using the provided arguments to the `postProcess(..)` method callback.

Chapter 11. Annotation Support for Function Execution

11.1. Introduction

Spring Data Geode includes annotation support to simplify working with Geode Function Execution. Under-the-hood, the Apache Geode API provides classes to implement and register Geode Functions that are deployed on Geode servers, which may then be invoked by other peer member applications or remotely from cache clients.

Functions can execute in parallel, distributed among multiple Geode servers in the cluster, aggregating results with the map-reduce pattern that are sent back to the caller. Functions can also be targeted to run on a single server or Region. The Apache Geode API supports remote execution of Functions targeted using various predefined scopes: on Region, on members [in groups], on servers, etc. The implementation and execution of remote Functions, as with any RPC protocol, requires some boilerplate code.

Spring Data Geode, true to *Spring's* core value proposition, aims to hide the mechanics of remote Function execution and allow developers to focus on core POJO programming and business logic. To this end, *Spring Data Geode* introduces annotations to declaratively register public methods of a POJO class as Geode Functions along with the ability to invoke registered Functions [remotely] via annotated interfaces.

11.2. Implementation vs Execution

There are two separate concerns to address implementation and execution.

First is Function implementation (server-side), which must interact with the FunctionContext to access the invocation arguments, ResultsSender as well as other execution context information. The Function implementation typically accesses the Cache and/or Regions and is registered with the FunctionService under a unique Id.

A cache client application invoking a Function does not depend on the implementation. To invoke a Function, the application instantiates an Execution providing the Function ID, invocation arguments and the Function target, which defines its scope: Region, server, servers, member or members. If the Function produces a result, the invoker uses a ResultCollector to aggregate and acquire the execution results. In certain cases, 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 Geode's client-server topology. While it is common for an application using a `ClientCache` to invoke a Function on one or more Geode servers in a cluster, it is also possible to execute Functions in a peer-to-peer (P2P) configuration, where the application is a member of the cluster hosting a peer `Cache`. Keep in mind that a peer member cache application is subject to all the same constraints of being a peer member of the cluster.

11.3. Implementing a Function

Using Geode APIs, the `FunctionContext` provides a runtime invocation context that includes the client's calling arguments and a `ResultSender` implementation to send results back to the client. Additionally, if the Function is executed on a Region, the `FunctionContext` is actually an instance of `RegionFunctionContext`, which provides additional information such as the target Region on which the Function was invoked and any Filter (set of specific keys) associated with the `Execution`, etc. 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 use 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 client's execution arguments. However, in the case of a Region execution, the Region data may 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 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 for the client and server to share a common interface, but this is not strictly 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 in the `FunctionContext` as an array:

```
Object[] args = new Object[] { "test", 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 context);
```

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 Function method parameters. The method's return type must be void or a type that may be serialized (either as a `java.io.Serializable`, `DataSerializable` or `PdxSerializable`). 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.

11.3.1. Annotations for Function Implementation

The following example illustrates how SDG's Function annotations are used to expose POJO methods as GemFire Functions:

```
@Component
public class ApplicationFunctions {

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

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

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

}
```

Note, the class itself must be registered as a *Spring* bean and each Geode Function is annotated with `@GemfireFunction`. In this example, *Spring's* `@Component` annotation was used, but you may register the bean by any method supported by *Spring* (e.g. XML configuration or with a Java configuration class using *Spring Boot*). This allows the *Spring* container to create an instance of this class and wrap it in a PojoFunctionWrapper. *Spring* creates a wrapper instance for each method annotated with `@GemfireFunction`. Each wrapper instance shares the same target object instance to invoke the corresponding method.

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

Spring creates the wrapper class and registers the Function(s) with Geode's Function Service. The Function id used to register the Functions must be unique. Using convention it defaults to the simple (unqualified) method name. The name can be explicitly defined using the `id` attribute of the `@GemfireFunction` annotation. The `@GemfireFunction` annotation also provides other configuration attributes, `HA` and `optimizedForWrite`, which correspond to properties defined by Geode's Function interface. If the method's return type is void, then the `hasResult` property is automatically set to `false`; otherwise, if the method returns a value the `hasResult` attributes is set to `true`.

Even for `void` return types, the annotation's `hasResult` attribute 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 `PojoFunctionWrapper` implements Geode's `Function` interface, binds method parameters and invokes the target method in its `execute()` method. It also sends the method's return value using the `ResultSender`.

11.3.2. Batching Results

If the return type is an array or Collection, then some consideration must be given to how the results are returned. By default, the `PojoFunctionWrapper` returns the entire array or Collection at once. If the number of elements in the array or Collection quite is large, it may incur a performance penalty. To divide the payload into smaller, more maneable chunks, you can set the `batchSize` attribute, as illustrated in `function2`, above.

TIP	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` and use it directly within the method to sends results back to the caller.

11.3.3. Enabling Annotation Processing

In accordance with *Spring* standards, you must explicitly activate annotation processing for `@GemfireFunction` annotations.

Using XML:

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

Or by annotating a Java configuration class:

```
@Configuration
@EnableGemfireFunctions
class ApplicationConfiguration { .. }
```

11.4. Executing a Function

A process invoking a remote Function needs to provide the Function's ID, calling arguments, the execution target (onRegion, onServers, onServer, onMember, onMembers) and optionally, a Filter set. Using *Spring Data Geode*, 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 the defined return type, if necessary. This technique is very similar to the way *Spring Data Geode's Repository extension* works, 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.

11.4.1. Annotations for Function Execution

To support client-side Function execution, the following SDG Function annotations are provided: `@OnRegion`, `@OnServer`, `@OnServers`, `@OnMember`, `@OnMembers`. These annotations correspond to the `Execution` implementations prodided by Geode's FunctionService. Each annotation exposes the appropriate attributes. These annotations also provide an optional `resultCollector` attribute whose

value is the name of a *Spring* bean implementing the ResultCollector to use for the execution.

CAUTION	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 a few 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. The `@FunctionId` annotation can be used to bind this invocation to a different Function ID.

11.4.2. Enabling Annotation Processing

The client-side uses *Spring's* classpath component scanning capability to discover annotated interfaces. To enable Function execution annotation processing in XML:

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

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 documentation.

Optionally, a developer can annotate her Java configuration class:

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

11.5. Programmatic Function Execution

Using the Function execution annotated interface defined in the previous section, simply auto-wire your interface into an application bean that will invoke the Function:

```
@Component
public class MyApplication {

    @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`.

Example 15. Using the `GemfireOnRegionFunctionTemplate`

```
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` as is. The first parameter is the Function ID. The Filter argument is optional. The following arguments are a variable argument `List`.

11.6. Function Execution with PDX

When using *Spring Data Geode's* Function annotation support combined with Apache Geode'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 Geode Functions using POJO classes annotated with Spring Data Geode Function annotations like so...

```
public class OrderFunctions {

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

}
```

NOTE	The Integer type, count parameter is arbitrary as is the separation of the `Order` class 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' Geode 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 with a `ClientCache` (i.e. `<gfe:client-cache/>`) based application. This implies that the Function arguments must also be serializable. The same is true when invoking peer-to-peer member Functions (e.g. `@OnMember(s)) between peers in the cluster. Any form of `distribution` requires the data transmitted between client and server, or peers, to be serialized.

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

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

Or from a Geode cache client application...

```
<gfe:client-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, including, but not limited to, Function arguments.

Geode will only serialize application domain object types that you have specifically configured (registered), with either Geode's ReflectionBasedAutoSerializer, or specifically (and recommended) using a "custom" Geode PdxSerializer. If you are using *Spring Data Geode's* Repository extension to *Spring Data Common's* Repository abstraction and infrastructure, you might even want to consider using *Spring Data Geode's* MappingPdxSerializer, which uses a entity's mapping meta-data to determine data from the application domain object that will be serialized to the PDX instance.

What is less than apparent, though, is that Geode automatically handles Java Enum types regardless of whether they are explicitly configured or not (i.e. registered with a `ReflectionBasedAutoSerializer` using a regex pattern and the `classes` parameter, or are handled by a "custom" Geode `PdxSerializer`), despite the fact that Java Enums implement `java.io.Serializable`.

So, when a developer sets `pdx-read-serialized` to **true** on Geode Servers where the Geode Functions (including Spring Data Geode Function annotated POJO classes) are registered, 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, what the Geode Function on the Server gets is...

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

The `Order` and `OrderSource` have been passed to the Function as PDX instances. Again, this is all because `pdx-read-serialized` is set to **true**, which may be necessary in cases where the Geode Servers are interacting with multiple different clients (e.g. Java, native clients, such as C++/C#, etc).

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

So, *Spring Data Geode* includes enhanced Function support to automatically convert method arguments passed to the Function that are of type PDX to the desired application domain object types defined by the Function method's parameter types.

However, this also requires the developer to explicitly register a Geode `PdxSerializer` on the Geode Servers where *Spring Data Geode* Function annotated POJOs are registered and used, e.g. ...

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

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

Alternatively, a developer my use Geode's ReflectionBasedAutoSerializer for convenience. Of course, it is recommended that you use a "custom" `PdxSerializer` where possible to maintain finer grained control over your serialization strategy.

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

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

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

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

Chapter 12. Apache Lucene Integration

Apache Geode integrates with Apache Lucene to allow developers to index and search on data stored in Apache Geode using Lucene queries. Search-based queries also includes the capability to page through query results.

Additionally, *Spring Data Geode* adds support for query projections based on *Spring Data Commons* Projection infrastructure. This feature enables the query results to be projected into first-class, application domain types as needed or required by the application use case.

However, a Lucene `Index` must be created first before any Lucene search-based query can be ran. A `LuceneIndex` can be created in *Spring (Data GemFire)* XML config like so...

```
<gfe:lucene-index id="IndexOne" fields="fieldOne, fieldTwo" region-path="/Example"/>
```

Additionally, Apache Lucene allows the specification of Analyzers per field and can be configured using...

```
<gfe:lucene-index id="IndexTwo" lucene-service-ref="luceneService" region-path=
"/AnotherExample">
    <gfe:field-analyzers>
        <map>
            <entry key="fieldOne">
                <bean class="example.AnalyzerOne"/>
             </entry>
            <entry key="fieldTwo">
                <bean class="example.AnalyzerTwo"/>
             </entry>
        </map>
    </gfe:field-analyzers>
</gfe:lucene-index>
```

Of course, the `Map` can be specified as a top-level bean definition and referenced using the `ref` attribute on the nested `<gfe:field-analyzers>` element like this, `<gfe-field-analyzers ref="refToTopLevelMapBeanDefinition"/>`.

Alternatively, a `LuceneIndex` can be declared in *Spring* Java config, inside a `@Configuration` class with...

```
@Bean(name = "People")
@DependsOn("personTitleIndex")
PartitionedRegionFactoryBean<Long, Person> peopleRegion(GemFireCache gemfireCache) {
    PartitionedRegionFactoryBean<Long, Person> peopleRegion = new
PartitionedRegionFactoryBean<>();

    peopleRegion.setCache(gemfireCache);
    peopleRegion.setClose(false);
    peopleRegion.setPersistent(false);

    return peopleRegion;
}

@Bean
LuceneIndexFactoryBean personTitleIndex(GemFireCache gemFireCache) {
    LuceneIndexFactoryBean luceneIndex = new LuceneIndexFactoryBean();

    luceneIndex.setCache(gemFireCache);
    luceneIndex.setFields("title");
    luceneIndex.setRegionPath("/People");

    return luceneIndex;
}
```

There are a few limitations of Apache Geode's, Apache Lucene integration support. First, a `LuceneIndex` can only be created on a Geode `PARTITION` Region. Second, all `LuceneIndexes` must be created before the the Region on which the `LuceneIndex` is applied.

It is possible that these Apache Geode restrictions will not apply in a future release which is why the SDG `LuceneIndexFactoryBean` API takes a reference to the Region directly as well, rather than just the Region path.

This is more ideal if think about the case in which users may want to define a `LuceneIndex` on an existing Region with data at a later point during the application's lifecycle and as requirements demand. Where possible, SDG strives to stick to strongly-typed objects.

Now that we have a `LuceneIndex` we can perform Lucene based data access operations, such as queries.

12.1. Lucene Template Data Accessors

Spring Data Geode provides 2 primary templates for Lucene data access operations, depending on how low a level your application is prepared to deal with.

The `LuceneOperations` interface defines query operations using Apache Geode Lucene types.

```
public interface LuceneOperations {

    <K, V> List<LuceneResultStruct<K, V>> query(String query, String defaultField [,
int resultLimit]
        , String... projectionFields);

    <K, V> PageableLuceneQueryResults<K, V> query(String query, String defaultField,
        int resultLimit, int pageSize, String... projectionFields);

    <K, V> List<LuceneResultStruct<K, V>> query(LuceneQueryProvider queryProvider [,
int resultLimit]
        , String... projectionFields);

    <K, V> PageableLuceneQueryResults<K, V> query(LuceneQueryProvider queryProvider,
        int resultLimit, int pageSize, String... projectionFields);

    <K> Collection<K> queryForKeys(String query, String defaultField [, int
resultLimit]);

    <K> Collection<K> queryForKeys(LuceneQueryProvider queryProvider [, int
resultLimit]);

    <V> Collection<V> queryForValues(String query, String defaultField [, int
resultLimit]);

    <V> Collection<V> queryForValues(LuceneQueryProvider queryProvider [, int
resultLimit]);
}
```

NOTE The `[, int resultLimit]` indicates that the `resultLimit` parameter is optional.

The operations in the `LuceneOperations` interface match the operations provided by the Apache Geode's LuceneQuery interface. However, SDG has the added value of translating proprietary Geode or Lucene `Exceptions` into *Spring's* highly consistent and expressive DAO Exception Hierarchy, particularly as many modern data access operations involve more than single store or repository.

Additionally, SDG's `LuceneOperations` interface can shield your application from interface breaking changes introduced by the underlying Apache Geode or Apache Lucene APIs when they do and will occur.

However, it would be remorse to only offer a Lucene Data Access Object that only uses Apache Geode and Apache Lucene data types (e.g. Geode's `LuceneResultStruct`), therefore SDG gives you the `ProjectingLuceneOperations` interface to remedy these important application concerns.

```
public interface ProjectingLuceneOperations {

    <T> List<T> query(String query, String defaultField [, int resultLimit], Class<T>
projectionType);

    <T> Page<T> query(String query, String defaultField, int resultLimit, int
pageSize, Class<T> projectionType);

    <T> List<T> query(LuceneQueryProvider queryProvider [, int resultLimit], Class<T>
projectionType);

    <T> Page<T> query(LuceneQueryProvider queryProvider, int resultLimit, int
pageSize, Class<T> projectionType);
}
```

The `ProjectingLuceneOperations` interface primarily uses application domain object types to work with your application data. The `query` method variants accept a projection type and the template applies the query results to instances of the given projection type using the *Spring Data Commons* Projection infrastructure.

Additionally, the template wraps the paged Lucene query results in an instance of the *Spring Data Commons* abstraction representing a `Page`. The same projection logic can still be applied to the results in the page and are lazily projected as each page in the collection is accessed.

By way of example, suppose I have a class representing a `Person` like so…

```
class Person {

    Gender gender;

    LocalDate birthDate;

    String firstName;
    String lastName;

    ...

    String getName() {
        return String.format("%1$s %2$s", getFirstName(), getLastName());
    }
}
```

Additionally, I might have a single interface to represent people as `Customers` depending on my application view…

```
interface Customer {

    String getName()
}
```

If I define the following `LuceneIndex`...

```
@Bean
LuceneIndexFactoryBean personLastNameIndex(GemFireCache gemfireCache) {
    LuceneIndexFactoryBean personLastNameIndex = new LuceneIndexFactoryBean();

    personLastNameIndex.setCache(gemfireCache);
    personLastNameIndex.setFields("lastName");
    personLastNameIndex.setRegionPath("/People");

    return personLastNameIndex;
}
```

Then it is a simple matter to query for people as either `Person` objects...

```
List<Person> people = luceneTemplate.query("lastName: D*", "lastName", Person.class);
```

Or as a `Page` of type `Customer`...

```
Page<Customer> customers = luceneTemplate.query("lastName: D*", "lastName", 100, 20,
Customer.class);
```

The `Page` can then be used to fetch individual pages of results...

```
List<Customer> firstPage = customers.getContent();
```

Conveniently, the *Spring Data Commons* `Page` interface implements `java.lang.Iterable<T>` too making it very easy to iterate over the content as well.

The only restriction to the *Spring Data Commons* Projection infrastructure is that the projection type must be an interface. However, it is possible to extend the provided, out-of-the-box (OOTB) SDC Projection infrastructure and provide a custom ProjectionFactory that uses CGLIB to generate proxy classes as the projected entity.

A custom `ProjectionFactory` can be set on a Lucene template using `setProjectionFactory(:ProjectionFactory)`.

12.2. Annotation configuration support

Finally, *Spring Data Geode* provides Annotation configuration support for `LuceneIndexes`. Eventually, the SDG Lucene support will find its way into the *Repository* infrastructure extension for Apache Geode so that Lucene queries can be expressed as methods on an application `Repository` interface, much like the OQL support today.

However, in the meantime, if you want to conveniently express `LuceneIndexes`, you can do so directly on your application domain objects like so...

```
@PartitionRegion("People")
class Person {

    Gender gender;

    @Index
    LocalDate birthDate;

    String firstName;

    @LuceneIndex;
    String lastName;

    ...
}
```

You must be using the SDG Annotation configuration support along with the `@EnableEntityDefineRegions` and `@EnableIndexing` Annotations to enable this feature...

```
@PeerCacheApplication
@EnableEntityDefinedRegions
@EnableIndexing
class ApplicationConfiguration {

  ...
}
```

Given our definition of the `Person` class above, the SDG Annotation configuration support will find the `Person` entity class definition, determine that people will be stored in a `PARTITION` Region called "People" and that the Person will have an OQL `Index` on `birthDate` along with a `LuceneIndex` on `lastName`.

More will be described with this feature in subsequent releases.

Chapter 13. Bootstrapping a Spring ApplicationContext in Apache Geode

13.1. Introduction

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

However, it is sometimes necessary, perhaps a requirement imposed by your IT organization, that Geode be fully managed and operated using the provided Apache Geode tool suite, such as with Gfsh. By using *Gfsh*, Geode will bootstrap your *Spring* application context rather than the other way around. Instead of an application server, or a Java main class using *Spring Boot*, whatever, Geode does the bootstrapping and will host your application.

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

13.2. Using Apache Geode to Bootstrap a Spring Context Started with Gfsh

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

An Initializer is declared within an initializer element using a minimal snippet of Geode's native `cache.xml`. 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="⋯"/>`)

Fortunately, such an Initializer is already conveniently provided by the framework, the SpringContextBootstrappingInitializer. A typical, yet very minimal configuration for this class inside Geodes's `cache.xml` file will look like this:

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

  <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 used to bootstrap a *Spring* application 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 packages containing appropriately annotated application components that the *Spring* container will search in order to find and create *Spring* beans and other application components on the classpath:

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

  <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 `cache.xml` file shown above, specified as a command-line option when starting a Geode Server in *Gfsh*, the command-line would

be:

```
gfsh>start server --name=Server1 --log-level=config ...
    --classpath="/path/to/application/classes.jar:/path/to/spring-data-geode
-<major>.<minor>.<maint>.RELEASE.jar"
    --cache-xml-file="/path/to/geode/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 a GemFire cache cannot be configured using the *Spring Data Geode* 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 reason for this is that Geode 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.

13.3. Lazy-Wiring GemFire Components

Spring Data Geode already provides existing support for wiring Geode components, such as `CacheListeners`, `CacheLoaders`, `CacheWriters` and so on, that are declared and created by Geode in `cache.xml` using SDG's `WiringDeclarableSupport` class as described in Configuration using auto-wiring and annotations. However, this only works when *Spring* is the one doing the bootstrapping (i.e. bootstrapping Geode).

When your *Spring* application context is bootstrapped by Geode, then these Geode application components go unnoticed since the *Spring* application context does not even exist yet! The *Spring* application context will not get created until Geode calls the Initializer block, which only occurs after all the other Geode 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* container that will eventually be created by Geode once the Initializer is called. In essence, this give your Geode defined application components a chance to be configured and auto-wired with *Spring* beans defined in the *Spring* application context.

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

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

  @Autowired
  private DataSource userDataSource;

  ...
}
```

As implied in the `CacheLoader` example above, you might necessarily (although, rarely) have defined both a Region and `CacheListener` component in Geode `cache.xml`. The `CacheLoader` may need access to an application DAO, or perhaps a *Spring* application context defined JDBC `DataSource` for loading `Users` into a Geode `REPLICATE` Region on start.

CAUTION Be careful when mixing the different life-cycles of Apache Geode and the *Spring* Container together in this manner as not all use cases and scenarios are supported. The Geode `cache.xml` configuration would be similar to the following (which comes from SDG's test suite):

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

  <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.SpringContextBootstrappingInitializerIntegrat
ionTest$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 14. Sample Applications

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

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

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

14.1. Hello World

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

NOTE	*Running under Linux* If you experience networking problems when starting Geode or the samples, try adding the following system property `java.net.preferIPv4Stack=true` to the command line (e.g. `-Djava.net.preferIPv4Stack=true`). For an alternative (global) fix especially on Ubuntu see SGF-28.

14.1.1. 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`. A developer 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 JVM and shutdown the *Spring* container.

14.1.2. 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 ran 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.

14.1.3. Hello World Sample Explained

Hello World uses both *Spring* XML and annotations for its configuration. The initial bootstrapping configuration is `app-context.xml`, which includes the cache configuration defined in the `cache-context.xml` file and performs classpath component scanning for *Spring* components.

The cache configuration defines the GemFire cache, Region and for illustrative purposes, a simple `CacheListener` 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.

Resources

In addition to this reference documentation, there are a number of other resources that may help you learn how to use Apache Geode with the *Spring Framework*. These additional, third-party resources are enumerated in this section.

Chapter 15. Useful Links

- Spring Data GemFire Project Page
- Spring Data Geode source code
- Spring Data Geode JIRA
- Spring Data GemFire on StackOverflow
- Archive of the Spring Data GemFire Forum on Spring IO
- Apache Geode Home Page
- Apache Geode Documentation
- Apache Geode Community
- Apache Geode source code
- Apache Geode JIRA
- Apache Geode on StackOverflow

Appendices

Appendix A: Namespace reference

The <repositories /> element

The `<repositories />` element triggers the setup of the Spring Data repository infrastructure. The most important attribute is `base-package` which defines the package to scan for Spring Data repository interfaces. [1: see [repositories.create-instances.spring]]

Table 6. Attributes

Name	Description
`base-package`	Defines the package to be used to be scanned for repository interfaces extending *Repository (actual interface is determined by specific Spring Data module) in auto detection mode. All packages below the configured package will be scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix will be considered as candidates. Defaults to `Impl`.
`query-lookup-strategy`	Determines the strategy to be used to create finder queries. See [repositories.query-methods.query-lookup-strategies] for details. Defaults to `create-if-not-found`.
`named-queries-location`	Defines the location to look for a Properties file containing externally defined queries.
`consider-nested-repositories`	Controls whether nested repository interface definitions should be considered. Defaults to `false`.

Appendix B: Populators namespace reference

The <populator /> element

The `<populator />` element allows to populate the a data store via the Spring Data repository infrastructure. [2: see [repositories.create-instances.spring]]

Table 7. Attributes

Name	Description
`locations`	Where to find the files to read the objects from the repository shall be populated with.

Appendix C: Repository query keywords

Supported query keywords

The following table lists the keywords generally supported by the Spring Data repository query derivation mechanism. However, consult the store-specific documentation for the exact list of supported keywords, because some listed here might not be supported in a particular store.

Table 8. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`
`AFTER`	`After`, `IsAfter`
`BEFORE`	`Before`, `IsBefore`
`CONTAINING`	`Containing`, `IsContaining`, `Contains`
`BETWEEN`	`Between`, `IsBetween`
`ENDING_WITH`	`EndingWith`, `IsEndingWith`, `EndsWith`
`EXISTS`	`Exists`
`FALSE`	`False`, `IsFalse`
`GREATER_THAN`	`GreaterThan`, `IsGreaterThan`
`GREATER_THAN_EQUALS`	`GreaterThanEqual`, `IsGreaterThanEqual`
`IN`	`In`, `IsIn`
`IS`	`Is`, `Equals`, (or no keyword)
`IS_EMPTY`	`IsEmpty`, `Empty`
`IS_NOT_EMPTY`	`IsNotEmpty`, `NotEmpty`
`IS_NOT_NULL`	`NotNull`, `IsNotNull`
`IS_NULL`	`Null`, `IsNull`
`LESS_THAN`	`LessThan`, `IsLessThan`
`LESS_THAN_EQUAL`	`LessThanEqual`, `IsLessThanEqual`
`LIKE`	`Like`, `IsLike`
`NEAR`	`Near`, `IsNear`
`NOT`	`Not`, `IsNot`
`NOT_IN`	`NotIn`, `IsNotIn`
`NOT_LIKE`	`NotLike`, `IsNotLike`
`REGEX`	`Regex`, `MatchesRegex`, `Matches`
`STARTING_WITH`	`StartingWith`, `IsStartingWith`, `StartsWith`
`TRUE`	`True`, `IsTrue`
`WITHIN`	`Within`, `IsWithin`

Appendix D: Repository query return types

Supported query return types

The following table lists the return types generally supported by Spring Data repositories. However, consult the store-specific documentation for the exact list of supported return types, because some listed here might not be supported in a particular store.

NOTE Geospatial types like (`GeoResult`, `GeoResults`, `GeoPage`) are only available for data stores that support geospatial queries.

Table 9. Query return types

Return type	Description
`void`	Denotes no return value.
Primitives	Java primitives.
Wrapper types	Java wrapper types.
`T`	An unique entity. Expects the query method to return one result at most. In case no result is found `null` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Iterator<T>`	An `Iterator`.
`Collection<T>`	A `Collection`.
`List<T>`	A `List`.
`Optional<T>`	A Java 8 or Guava `Optional`. Expects the query method to return one result at most. In case no result is found `Optional.empty()` /`Optional.absent()` is returned. More than one result will trigger an `IncorrectResultSizeDataAccessException`.
`Option<T>`	An either Scala or JavaSlang `Option` type. Semantically same behavior as Java 8's `Optional` described above.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects method to be annotated with `@Async` and requires Spring's asynchronous method execution capability enabled.
`Slice`	A sized chunk of data with information whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, e.g. the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, e.g. distance to a reference location.

Return type	Description
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, e.g. average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, e.g. average distance to a reference location.

Appendix E: Spring Data Geode Schema

- Spring Data for Apache Geode Core Schema (`gfe`-namespace)
- Spring Data for Apache Geode Data Access Schema (`gfe-data`-namespace)