Spring Data for Pivotal GemFire Reference Guide

Costin Leau · David Turanski · John Blum · Oliver Gierke · Jay Bryant – Version 2.1.1.RELEASE, 2018-10-15

Table of Contents

Preface

Spring Data for Pivotal GemFire focuses on integrating the Spring Framework's powerful, non-invasive programming model and concepts with Pivotal GemFire to simplify configuration and development of Java applications when using Pivotal GemFire as you data management solution.

This document assumes you already have a basic understanding of, and some familiarity with, the core Spring Framework and Pivotal GemFire concepts.

While every effort has been made to ensure this documentation is comprehensive and complete without errors, some topics are beyond the scope of this document and may require more explanation (for example, data distribution management using partitioning with HA while still preserving consistency). Additionally, some typographical errors might have crept in. If you do spot mistakes or even more serious errors, please bring these issues to the attention of the Spring Data team by raising an appropriate issue in JIRA.

1. Introduction

The Spring Data for Pivotal GemFire reference guide explains how to use the Spring Framework to configure and develop applications with Pivotal GemFire. It presents the basic concepts and provides numerous examples to help you get started quickly.

2. Requirements

Spring Data for Pivotal GemFire requires Java 8.0, Spring Framework 5 and Pivotal GemFire 9.5.1.

3. New Features

As of the 1.2.0.RELEASE, this project, formerly known as Spring GemFire, has been renamed to Spring Data for Pivotal GemFire to reflect that it is now a module of the Spring Data project and built on Pivotal GemFire.

3.1. New in the 1.2 Release

- Full support for Pivotal GemFire configuration through the SDG `gfe` XML namespace. Now Pivotal GemFire components may be configured completely without requiring a native `cache.xml` file.
- WAN Gateway support for Pivotal GemFire 6.6.x. See Configuring WAN Gateways
- Spring Data Repository support using a dedicated SDG XML namespace, **gfe-data**. See Spring Data for Pivotal GemFire Repositories

- `gfe-data` XML Namespace support for registering Pivotal GemFire Functions. See Configuring the Function Service
- A top-level `<disk-store>` element has been added to the SDG `gfe` XML namespace to allow sharing of persist stores among regions as well as other Pivotal GemFire components that support persistent backup or overflow. See [bootstrap-diskstore]

`<*-region>` elements no longer allow a nested `<disk-store>` element.

- Pivotal GemFire Sub-Regions are supported by nested `<*-region>` elements.
- A `<local-region>` element has been added to configure a Local Region.
- Support for the re-designed WAN Gateway in Pivotal GemFire 7.0.

3.2. New in the 1.3 Release

- Upgraded to Spring Framework 3.2.8.
- Upgraded to Spring Data Commons 1.7.1.
- Annotation support for Pivotal GemFire Functions. It is now possible to declare and register Functions written as POJOs by using annotations. In addition, Function executions are defined as annotated interfaces, similar to the way Spring Data Repositories work. See Annotation Support for Function Execution.
- Added a `<datasource>` element to the SDG XML namespace to simplify establishing a basic client connection to a Pivotal GemFire data grid.
- Added a `<json-region-autoproxy>` element to the SDG `gfe-data` XML namespace to support JSON features introduced in Pivotal GemFire 7.0, enabling Spring AOP to perform the necessary conversions automatically on Region data access operations.
- Upgraded to Pivotal GemFire 7.0.1 and added XML namespace support for new `AsyncEventQueue` attributes.
- Added support for setting subscription interest policy on Regions.
- Support for void returns on Function executions. See Annotation Support for Function Execution for complete details.
- Support for persisting Local Regions. See Local Region.
- Support for entry time-to-live (TTL) and entry idle-time (TTI) on a Pivotal GemFire Client Cache. See Configuring a Pivotal GemFire ClientCache
- Support for multiple Spring Data for Pivotal GemFire web-based applications by using a single Pivotal GemFire cluster, operating concurrently inside tc Server.
- Support for `concurrency-checks-enabled` on all Cache Region definitions by using the SDG `gfe` XML namespace. See [bootstrap:region:common:attributes]
- Support for `CacheLoaders` and `CacheWriters` on client, Local Regions.
- Support for registering `CacheListeners`, `AsyncEventQueues`, and `GatewaySenders` on Pivotal GemFire Cache Sub-Regions.
- Support for PDX persistent keys in Regions.
- Support for correct Partition Region bean creation in a Spring context when collocation is specified with the `colocated-with` attribute.
- Full support for Cache Sub-Regions using proper, nested `<*-region>` element syntax in the SDG `gfe` XML namespace.

3.3. New in the 1.4 Release

- Upgraded to Pivotal GemFire 7.0.2.
- Upgraded to Spring Framework 3.2.13.RELEASE.
- Upgraded to Spring Data Commons 1.8.6.RELEASE.
- Integrated Spring Data for Pivotal GemFire with Spring Boot, which includes both a `spring-boot-starter-data-gemfire` POM and a Spring Boot sample application that demonstrates Pivotal GemFire Cache Transactions configured with SDG and bootstrapped with Spring Boot.
- Added support for bootstrapping a Spring `ApplicationContext` in a Pivotal GemFire Server when started from `Gfsh`. See Bootstrapping a Spring ApplicationContext in Pivotal GemFire
- Added support for persisting application domain object and entities to multiple Pivotal GemFire Cache Regions. See Entity Mapping
- Added support for persisting application domain object and entities to Pivotal GemFire Cache Sub-Regions, avoiding collisions when Sub-Regions are uniquely identifiable, but identically named. See Entity Mapping
- Added strict XSD type rules to Data Policies and Region Shortcuts on all Pivotal GemFire Cache Region types.
- Changed the default behavior of SDG `<*-region>` elements from lookup to always create a new Region with an option to restore the old behavior using the `ignore-if-exists` attribute. See Common Region Attributes and [bootstrap:region:common:regions-subregions-lookups-caution]
- Spring Data for Pivotal GemFire can now be fully built and run on JDK 7 and JDK 8.

3.4. New in the 1.5 Release

- Maintained compatibility with Pivotal GemFire 7.0.2.
- Upgraded to *Spring Framework* 4.0.9.RELEASE.
- Upgraded to *Spring Data Commons* 1.9.4.RELEASE.
- Converted Reference Guide to Asciidoc.
- Renewed support for deploying Spring Data for Pivotal GemFire in an OSGi container.
- Removed all default values specified in Spring Data for Pivotal GemFire XML namespace Region-type elements to rely on Pivotal GemFire defaults instead.
- Added convenience to automatically create `DiskStore` directory locations.
- SDG annotated Function implementations can now be executed from `Gfsh`.
- Enabled Pivotal GemFire `GatewayReceivers` to be started manually.
- Added support for Auto Region Lookups. See [bootstrap:region:auto-lookup]
- Added support for Region Templates. See [bootstrap:region:common:region-templates]

3.5. New in the 1.6 Release

- Upgraded to Pivotal GemFire 8.0.0.
- Maintained compatibility with Spring Framework 4.0.9.RELEASE.
- Upgraded to Spring Data Commons 1.10.2.RELEASE.
- Added support for Pivotal GemFire 8's new Cluster-based Configuration Service.
- Enabled 'auto-reconnect' functionality to be employed in Spring-configured Pivotal GemFire Servers.

- Allowed the creation of concurrent and parallel `AsyncEventQueues` and `GatewaySenders`.
- Added support for Pivotal GemFire 8's Region data compression.
- Added attributes to set both critical and warning percentages on `DiskStore` usage.
- Supported the capability to add `EventSubstitutionFilters` to `GatewaySenders`.

3.6. New in the 1.7 Release

- Upgraded to Pivotal GemFire 8.1.0.
- Upgraded to Spring Framework 4.1.9.RELEASE.
- Upgraded to Spring Data Commons 1.11.6.RELEASE.
- Added early access support for Apache Geode.
- Added support for adding Spring-defined `CacheListeners`, `CacheLoaders`, and `CacheWriters` on existing Regions configured in Spring XML, `cache.xml`, or even with Pivotal GemFire's Cluster Configuration Service.
- Added Spring JavaConfig support to `SpringContextBootstrappingInitializer`.
- Added support for custom `ClassLoaders` in `SpringContextBootstrappingInitializer` to load Spring-defined bean classes.
- Added support for `LazyWiringDeclarableSupport` re-initialization and complete replacement for `WiringDeclarableSupport`.
- Added `locators` and `servers` attributes to the `<gfe:pool>` element, allowing variable Locator and Server endpoint lists configured with Spring's property placeholders.
- Enables the use of the `<gfe-data:datasource>` element with non-Spring-configured Pivotal GemFire Servers.
- Added multi-index definition and creation support.
- Annotation-based Data Expiration
- [gemfire-repositories:oql-extensions]
- Added support for Cache and Region data snapshots. See Configuring the Snapshot Service

3.7. New in the 1.8 Release

- Upgraded to Pivotal GemFire 8.2.0.
- Upgraded to Spring Framework 4.2.9.RELEASE.
- Upgraded to Spring Data Commons 1.12.11.RELEASE.
- Added Maven POM to build SDG with Maven.
- Added support for CDI.
- Enabled a `ClientCache` to be configured without a `Pool`.
- Defaulted `<gfe:cache>` and `<gfe:client-cache>` elements `use-bean-factory-locator` attribute to **false**.
- Added `durable-client-id` and `durable-client-timeout` attributes to `<gfe:client-cache>`.
- Made `GemfirePersistentProperty` now properly handle other non-entity, scalar-like types (e.g. `BigDecimal` and `BigInteger`).
- Prevented SDG-defined `Pools` from being destroyed before `Regions` that use those `Pools`.
- Handled case-insensitive Pivotal GemFire OQL queries defined as Repository query methods.
- Changed `GemFireCache.evict(key)` to call `Region.remove(key)` in SDG's Spring *Cache Abstraction* support.

- Fixed `RegionNotFoundException` with Repository queries on a client `Region` associated with a specific `Pool` configured for Pivotal GemFire server groups.
- Changed `GatewaySenders/Receivers` to no longer be tied to the Spring container.

3.8. New in the 1.9 Release

- Upgraded to Pivotal GemFire 8.2.11.
- Upgraded to Spring Framework 4.3.18.RELEASE.
- Upgraded to Spring Data Commons 1.13.13.RELEASE.
- Introduced an entirely new Annotation-based configuration model inspired by Spring Boot.
- Added support for suspend and resume in the `GemfireTransactionManager`.
- Added support in Repositories to use the bean `id` property as the Region key when the `@Id` annotation is not present.
- Used `MappingPdxSerializer` as the default Pivotal GemFire serialization strategy when `@EnablePdx` is used.
- Enabled `GemfireCacheManager` to explicitly list Region names to be used in the Spring's *Caching Abstraction*.
- Configured Pivotal GemFire Caches, CacheServers, Locators, Pools, Regions, Indexes, DiskStores, Expiration, Eviction, Statistics, Mcast, HttpService, Auth, SSL, Logging, System Properties.
- Added Repository support with multiple Spring Data modules on the classpath.

3.9. New in the 2.0 Release

- Upgraded to Pivotal GemFire 9.1.1.
- Upgraded to Spring Data Commons 2.0.8.RELEASE.
- Upgraded to Spring Framework 5.0.7.RELEASE.
- Reorganized the SDG codebase by packaging different classes and components by concern.
- Added extensive support for Java 8 types, particularly in the SD Repository abstraction.
- Changed to the Repository interface and abstraction, e.g. IDs are no longer required to be `java.io.Serializable`.
- Set `@EnableEntityDefinedRegions` annotation `ignoreIfExists` attribute to `true` by default.
- Set `@Indexed` annotation `override` attribute to `false` by default.
- Renamed `@EnableIndexes` to `@EnableIndexing`.
- Introduced a `InterestsBuilder` class to easily and conveniently express Interests in keys and values between client and server when using JavaConfig.
- Added support in the Annotation configuration model for Off-Heap, Redis Adapter, and Pivotal GemFire's new Security framework.

3.10. New in the 2.1 Release

- Upgraded to Pivotal GemFire 9.5.1.
- Upgraded to Spring Framework 5.1.0.RELEASE.
- Upgraded to Spring Data Commons 2.1.0.RELEASE.
- Added support for parallel cache/Region snapshots along with invoking callbacks when loading snapshots.
- Added support for registering QueryPostProcessors to customize the OQL generated fro Repository query methods.
- Added support for include/exclude TypeFilters in o.s.d.g.mapping.MappingPdxSerializer.

- Updated docs.

Reference Guide

4. Document Structure

The following chapters explain the core functionality offered by Spring Data for Pivotal GemFire:

- Bootstrapping Pivotal GemFire with the Spring Container describes the configuration support provided for configuring, initializing, and accessing Pivotal GemFire Caches, Regions, and related distributed system components.
- Working with Pivotal GemFire APIs explains the integration between the Pivotal GemFire APIs and the various data access features available in Spring, such as template-based data access, exception translation, transaction management, and caching.
- Working with Pivotal GemFire Serialization describes enhancements to Pivotal GemFire's serialization and deserialization of managed objects.
- POJO Mapping describes persistence mapping for POJOs stored in Pivotal GemFire using Spring Data.
- Spring Data for Pivotal GemFire Repositories describes how to create and use Spring Data Repositories to access data stored in Pivotal GemFire by using basic CRUD and simple query operations.
- Annotation Support for Function Execution describes how to create and use Pivotal GemFire Functions by using annotations to perform distributed computations where the data lives.
- Continuous Query (CQ) describes how to use Pivotal GemFire's Continuous Query (CQ) functionality to process a stream of events based on interest that is defined and registered with Pivotal GemFire's OQL (Object Query Language).
- Bootstrapping a Spring ApplicationContext in Pivotal GemFire describes how to configure and bootstrap a Spring `ApplicationContext` running in an Pivotal GemFire server using `Gfsh`.
- Sample Applications describes the examples provided with the distribution to illustrate the various features available in Spring Data for Pivotal GemFire.

---

5. Bootstrapping Pivotal GemFire with the Spring Container

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

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

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

Spring Data for Pivotal GemFire's XML namespace supports full configuration of the Pivotal GemFire In-Memory Data Grid (IMDG). The XML namespace is one of two ways to configure Pivotal GemFire in a Spring context in order to properly

manage Pivotal GemFire's lifecycle inside the Spring container. The other way to configure Pivotal GemFire in a Spring context is by using annotation-based configuration.

While support for Pivotal GemFire's native `cache.xml` persists for legacy reasons, Pivotal GemFire application developers who use XML configuration 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 (Spring Expression Language), and environment profiles. Behind the XML namespace, Spring Data for Pivotal GemFire makes extensive use of Spring's `FactoryBean` pattern to simplify the creation, configuration, and initialization of Pivotal GemFire components.

Pivotal GemFire provides several callback interfaces, such as `CacheListener`, `CacheLoader`, and `CacheWriter`, that let developers add custom event handlers. Using Spring's IoC container, you can configure these callbacks as normal Spring beans and inject them into Pivotal GemFire components. This is a significant improvement over native `cache.xml`, which provides relatively limited configuration options and requires callbacks to implement Pivotal GemFire's `Declarable` interface (see Wiring `Declarable` Components to see how you can still use `Declarables` within Spring's 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.

5.2. Using the Core Namespace

To simplify configuration, Spring Data for Pivotal GemFire provides a dedicated XML namespace for configuring core Pivotal GemFire components. It is possible to configure beans directly by using Spring's standard `<bean>` definition. However, all bean properties are exposed through 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.

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

To use the Spring Data for Pivotal GemFire XML namespace, declare it in your Spring XML configuration meta-data, as the following example shows:

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

  <bean id ... >

  <gfe:cache ...>  4

</beans>
```

1. Spring Data for Pivotal GemFire XML namespace prefix. Any name works, but, throughout this reference documentation, `gfe` is 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 resolves the schema locally, as it is included in the Spring Data for Pivotal GemFire library.
4. Example declaration using the XML namespace with the `gfe` prefix.

You can change the default namespace from `beans` to `gfe`. This is useful for XML configuration composed mainly of Pivotal GemFire components, as it avoids declaring the prefix. To do so, swap the namespace prefix declaration shown earlier, as the following example shows:

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

  <beans:bean id ... >  3

  <cache ...>  4

</beans>
```

1. The default namespace declaration for this XML document points to the Spring Data for Pivotal GemFire 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 for Pivotal GemFire provides a data access XML namespace (`gfe-data`), which is primarily intended to simplify the development of Pivotal GemFire client applications. This namespace currently contains support for Pivotal GemFire Repositories and Function execution, as well as a `<datasource>` tag that offers a convenient way to connect to a Pivotal GemFire cluster.

5.3.1. An Easy Way to Connect to Pivotal GemFire

For many applications, a basic connection to a Pivotal GemFire data grid using default values is sufficient. Spring Data for Pivotal GemFire's `<datasource>` tag provides a simple way to access data. The data source creates a `ClientCache` and connection `Pool`. In addition, it queries the cluster servers for all existing root Regions and creates 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` elements to connect to an existing data grid. Additionally, all attributes available to configure a Pool are supported. This configuration automatically creates client Region beans for each Region defined on cluster members connected to the Locator, so they can be seamlessly referenced by Spring Data mapping annotations (`GemfireTemplate`) and autowired into application classes.

Of course, you can explicitly configure client Regions. For example, if you want to cache data in local memory, as the following example shows:

```
<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 Pivotal GemFire, you need to either create a new cache or connect to an existing one. With the current version of Pivotal GemFire, you can have only one open cache per VM (more strictly speaking, per `ClassLoader`). In most cases, the cache should only be created once.

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 stand-alone applications and integration tests. However, in typical production systems, most application processes act as cache clients, creating a `ClientCache` instance instead. This is described in the Configuring a Pivotal GemFire ClientCache and Client Region sections.

A peer `Cache` with default configuration can be created with the following simple declaration:

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

During Spring container initialization, any `ApplicationContext` containing this cache definition registers a `CacheFactoryBean` that creates a Spring bean named `gemfireCache`, which references a Pivotal GemFire `Cache` instance. This bean refers 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` applies the default cache configuration.

All Spring Data for Pivotal GemFire components that depend on the `Cache` respect this naming convention, so you need not explicitly declare the `Cache` dependency. If you prefer, you can make the dependency explicit by using the `cache-ref` attribute provided by various SDG XML namespace elements. Also, you can override the cache's bean name using the `id` attribute, as follows:

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

A Pivotal GemFire `Cache` can be fully configured using Spring. However, Pivotal GemFire's native XML configuration file, `cache.xml`, is also supported. For situations where the Pivotal GemFire cache needs to be configured natively, you can provide a reference to the Pivotal GemFire XML configuration file by using the `cache-xml-location` attribute, as follows:

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

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

The configuration makes use of Spring's `Resource` abstraction to locate the file. The `Resource` abstraction lets various search patterns 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, you can also specify Pivotal GemFire System properties that use any of Spring's `Properties` support features.

For example, you can use the `properties` element defined in the `util` namespace to define `Properties` directly or load properties from a properties file, 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:util="http://www.springframework.org/schema/util"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/gemfire http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
    http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd
">

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

Cache settings apply only when 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, as the following listing shows:

```
<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"
```

```
    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.gemfire.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 Attributes support various cache options. For further information regarding anything shown in this example, see the Pivotal GemFire 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` (the default is `false`) lets a disconnected Pivotal GemFire member automatically reconnect and rejoin the Pivotal GemFire cluster. See the Pivotal GemFire product documentation for more details.

3 Setting the `use-bean-factory-locator` attribute to `true` (it defaults to `false`) applies only when both Spring (XML) configuration metadata and Pivotal GemFire `cache.xml` is used to configure the Pivotal GemFire cache node (whether client or peer). This option lets Pivotal GemFire components (such as `CacheLoader`) expressed in `cache.xml` be auto-wired with beans (such as `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` (the default is `false`) enables a Pivotal GemFire member to retrieve the common, shared Cluster-based configuration from a Locator. See the Pivotal GemFire product documentation for more details.

5 Example of a `TransactionListener` callback declaration that uses a bean reference. The referenced bean must implement TransactionListener. A `TransactionListener` can be implemented to handle transaction related events (such as afterCommit and afterRollback).

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

7 Example of a `GatewayConflictResolver` callback declaration using a bean reference. The referenced bean must implement http://gemfire-95-javadocs.docs.pivotal.io//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 Enables Pivotal GemFire's DynamicRegionFactory, which provides a distributed Region creation service.

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

Enabling PDX Serialization

The preceding example includes a number of attributes related to Pivotal GemFire's enhanced serialization framework, PDX. While a complete discussion of PDX is beyond the scope of this reference guide, it is important to note that PDX is enabled by registering a `PdxSerializer`, which is specified by setting the `pdx-serializer` attribute.

Pivotal GemFire 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 Pivotal GemFire Serialization.

Enabling Auto-reconnect

You should be careful when setting the `<gfe:cache enable-auto-reconnect="[true|false*]>` attribute to `true`.

Generally, 'auto-reconnect' should only be enabled in cases where Spring Data for Pivotal GemFire's XML namespace is used to configure and bootstrap a new, non-application Pivotal GemFire server added to a cluster. In other words, 'auto-reconnect' should not be enabled when Spring Data for Pivotal GemFire is used to develop and build a Pivotal GemFire application that also happens to be a peer `Cache` member of the Pivotal GemFire cluster.

The main reason for this restriction is that most Pivotal GemFire applications use references to the Pivotal GemFire `Cache` or Regions in order to perform data access operations. These references are "injected" by the Spring container into application components (such as 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 shuts down and all Pivotal GemFire component references (caches, Regions, and others) 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 caches, Regions, and other Pivotal GemFire 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.

Pivotal GemFire makes no guarantee (even when using the Pivotal GemFire public Java API) that application cache, Regions, or other component references are automatically refreshed by the reconnect operation. As such, Pivotal GemFire 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 either. If that were the case, you would have a clean way to know when to call `ConfigurableApplicationContext.refresh()`, if it were even applicable for an application to do so, which is why this "feature" of Pivotal GemFire is not recommended for peer `Cache` applications.

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

Using Cluster-based Configuration

Pivotal GemFire'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 is compatible with the Pivotal GemFire Distributed System when the member joins.

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

All Pivotal GemFire native configuration metadata, whether from `cache.xml` or from the Cluster Configuration Service, gets applied before any Spring (XML) configuration metadata. As a result, Spring's config serves to "augment" the native Pivotal GemFire configuration metadata and would most likely be specific to the application.

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

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

While certain Pivotal GemFire tools, such as *Gfsh*, have their actions "recorded" when schema-like changes are made (for example, `gfsh>create region --name=Example --type=PARTITION`), Spring Data for Pivotal GemFire's configuration metadata is not recorded. The same is true when using Pivotal GemFire's public Java API directly. It, too, is not recorded.

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

5.4.2. Configuring a Pivotal GemFire CacheServer

Spring Data for Pivotal GemFire includes dedicated support for configuring a CacheServer, allowing complete configuration through the Spring container, as the following example shows:

```
<?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/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/context http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/gemfire http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
">

  <gfe:cache/>

  <!-- Example depicting serveral Pivotal GemFire CacheServer configuration options -->
  <gfe:cache-server id="advanced-config" auto-startup="true"
       bind-address="localhost" host-name-for-clients="localhost" port="${gemfire.cache.server.port}"
       load-poll-interval="2000" max-connections="22" max-message-count="1000" max-threads="16"
       max-time-between-pings="30000" groups="test-server">

    <gfe:subscription-config eviction-type="ENTRY" capacity="1000" disk-store="file://${java.io.tmpdir}"/>

  </gfe:cache-server>

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

</beans>
```

The preceding configuration shows the `cache-server` element and the many available options.

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

To avoid initialization problems, the `CacheServer` started by Spring Data for Pivotal GemFire starts **after** the Spring container has been fully initialized. Doing so lets potential Regions, listeners, writers or instantiators that are 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 before your components and thus not be seen by the clients connecting right away.

5.4.3. Configuring a Pivotal GemFire ClientCache

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

The simplest definition of a Pivotal GemFire cache client using default configuration follows:

```
<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` and 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 for 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, as the following example shows:

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

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

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

Client Region covers client-side configuration in more detail.

Pivotal GemFire's DEFAULT Pool and Spring Data for Pivotal GemFire Pool Definitions

If a Pivotal GemFire `ClientCache` is local-only, then no Pool definition is required. For instance, you can define the following:

```
<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 Pivotal GemFire’s `ClientRegionShortcut` (all `LOCAL_*` shortcuts).

However, if a client Region is a (caching) proxy to a server-side Region, a Pool is required. In that case, there are several ways to define and use a Pool.

When a `ClientCache`, a Pool, and a proxy-based Region are all defined but not explicitly identified, Spring Data for Pivotal GemFire resolves the references automatically, as the following example shows:

```
<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 preceding example, the `ClientCache` is identified as `gemfireCache`, the Pool as `gemfirePool`, and the client Region as “Example”. However, the `ClientCache` initializes Pivotal GemFire’s `DEFAULT` Pool from `gemfirePool`, and the client Region uses the `gemfirePool` when distributing data between the client and the server.

Basically, Spring Data for Pivotal GemFire resolves the preceding 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"/>
```

Pivotal GemFire still creates a Pool called `DEFAULT`. Spring Data for Pivotal GemFire causes the `DEFAULT` Pool to be initialized from the `gemfirePool`. Doing so is useful in situations where multiple Pools are defined and client Regions are using separate Pools, or do not declare a Pool at all.

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 Pivotal GemFire `client-cache` `DEFAULT` pool is initialized from `locatorPool`, as specified by the `pool-name` attribute. There is no Spring Data for Pivotal GemFire-defined `gemfirePool`, since both Pools were explicitly identified (named) — `locatorPool` and `serverPool`, respectively.

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

Finally, the `YetAnotherExample` Region does not use a Pool, because it is `LOCAL`.

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

If we either changed the name of `locatorPool` to `gemfirePool` or made the Pool bean definition be anonymous, it would have the same effect as the preceding 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.

Pivotal GemFire implements the following types of Regions:

- **REPLICATE** - Data is replicated across all cache members in the cluster 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 many cache members in the cluster 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 Pivotal GemFire's documentation on Region Types.

5.5.1. Using an externally configured Region

To reference Regions already configured in a Pivotal GemFire 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"/>
```

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` lets you retrieve existing, pre-configured Regions without exposing the Region semantics or setup infrastructure.

5.5.2. Auto Region Lookup

`auto-region-lookup` lets you import all Regions defined in a Pivotal GemFire native `cache.xml` file into a Spring `ApplicationContext` when you use the `cache-xml-location` attribute on the `<gfe:cache>` element.

For instance, consider the following `cache.xml` 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="Parent" refid="REPLICATE">
    <region name="Child" refid="REPLICATE"/>
  </region>

</cache>
```

You can import the preceding `cache.xml` file as follows:

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

You can then use the `<gfe:lookup-region>` element (for example, `<gfe:lookup-region id="Parent"/>`) to reference specific Regions as beans in the Spring container, or you can choose to import all Regions defined in `cache.xml` by using the following:

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

Spring Data for Pivotal GemFire automatically creates beans for all Pivotal GemFire Regions defined in `cache.xml` that have not been explicitly added to the Spring container with explicit `<gfe:lookup-region>` bean declarations.

It is important to realize that Spring Data for Pivotal GemFire uses a Spring BeanPostProcessor to post-process the cache after it is both created and initialized to determine the Regions defined in Pivotal GemFire to add as beans in the Spring `ApplicationContext`.

You may inject these "auto-looked-up" Regions as you would any other bean defined in the Spring `ApplicationContext`, with one 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 preceding example only applies when you use Spring's `component-scan` functionality.

If you declare your components by using Spring XML config, then you would do the following:

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

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

5.5.3. Configuring Regions

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

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

See the Pivotal GemFire documentation for a comprehensive description of Region Types.

Common Region Attributes

The following table lists the attributes available for all Region types:

Table 1. Common Region Attributes

Name	Values	Description
cache-ref	Pivotal GemFire Cache bean reference	The name of the bean defining the Pivotal GemFire 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 Pivotal GemFire's data policy.	The region's data policy. Note that 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 through the `disk-store` element.
disk-synchronous	boolean (default: `true`)	Determines whether disk store writes are synchronous.
id	Any valid bean name.	The default region name 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.

Name	Values	Description
ignore-jta	boolean (default: `false`)	Determines whether this Region participates in JTA (Java Transaction API) transactions.
index-update-type	`synchronous` or `asynchronous` (default: `synchronous`)	Determines whether Indices are 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.
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 assumes the value of the `id` attribute (that is, the bean name).
persistent	*boolean (default: `false`)	Determines whether the region persists entries to local disk (disk store).
shortcut	See http://gemfire-95-javadocs.docs.pivotal.io//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 through one of the `*region-template` elements.
value-constraint	Any valid, fully-qualified Java class name.	Expected value type.

`CacheListener` instances

`CacheListener` instances 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 with the `cache-listener` element nested in the containing `*-region` element.

The following example has two declared `CacheListener's`. The first references a named, top-level Spring bean. The second is an anonymous inner bean definition.

```
<bean id="myListener" class="org.example.app.geode.cache.SimpleCacheListener"/>

<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>
</gfe:replicated-region>
```

The following example uses an alternate form of the `cache-listener` element with the `ref` attribute. Doing so allows for more concise configuration when defining a single `CacheListener`.

Note: The XML namespace allows only a single `cache-listener` element, so either the style shown in the preceding example or the style in the following example must be used.

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

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

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 results in an exception.

Bean Reference Conventions

The `cache-listener` element is an example of a common pattern used in the XML namespace anywhere Pivotal GemFire provides a callback interface to be implemented in order to invoke custom code in response to cache or Region events. When you use 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, if multiple instances are permitted, it may contain nested bean references and inner bean definitions in any combination. The convention is to use the singular form (that is, `cache-listener` vs `cache-listeners`), reflecting that the most common scenario is, in fact, 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 XML namespace provides `cache-loader` and `cache-writer` elements to register these Pivotal GemFire components for a Region.

A `CacheLoader` is invoked on a cache miss to let an entry be loaded from an external data source, such as a database. A `CacheWriter` is invoked before an entry is created or updated, to allow the entry to be synchronized to an external data source. The main difference is that Pivotal GemFire supports, at most, a single instance of `CacheLoader` and `CacheWriter` per Region. However, either declaration style may be used.

The following example declares a Region with both a `CacheLoader` and a `CacheWriter`:

```
<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 Pivotal GemFire documentation for more details.

5.5.4. Compression

Pivotal GemFire Regions may also be compressed in order to reduce JVM memory consumption and pressure to possibly avoid global GCs. When you enable compression for a Region, all values stored in memory for the Region are compressed, while keys and indexes remain uncompressed. New values are compressed when put into the 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.

The following example shows a Region with compression enabled:

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

See Pivotal GemFire's documentation for more information on Region Compression.

5.5.5. Off-Heap

Pivotal GemFire Regions may also be configured to store Region values in off-heap memory, which is a portion of JVM memory that is not subject to Garbage Collection (GC). By avoid expensive GC cycles, your application can spend more of its time on things that matter, like processing requests.

Using off-heap memory is as simple as declaring the amount of memory to use and then enabling your Regions to use off-heap memory, as shown in the following configuration:

```
<util:properties id="gemfireProperties">
    <prop key="off-heap-memory-size">200G</prop>
</util:properties>

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

```
<gfe:partitioned-region id="ExampleOffHeapRegion" off-heap="true"/>
```

You can control other aspects of off-heap memory management by setting the following Pivotal GemFire configuration properties using the `<gfe:cache>` element:s

```
<gfe:cache critical-off-heap-percentage="90" eviction-off-heap-percentage"80"/>
```

Pivotal GemFire's `ResourceManager` will use these two threshold values (`critical-off-heap-percentage` & `eviction-off-heap-percentage`) to more effectively manage the off-heap memory in much the same way as the JVM does when managing heap memory. Pivotal GemFire `ResourceManager` will prevent the cache from consuming too much off-heap memory by evicting old data. If the off-heap manager is unable to keep up, then the `ResourceManager` refuses additions to the cache until the off-heap memory manager has freed up an adequate amount of memory.

See Pivotal GemFire's documentation for more information on Managing Heap and Off-Heap Memory.

Specifically, read the section, Managing Off-Heap Memory.

5.5.6. Subregions

Spring Data for Pivotal GemFire also supports Sub-Regions, allowing Regions to be arranged in a hierarchical relationship.

For example, Pivotal GemFire allows for a `/Customer/Address` Region and a different `/Employee/Address` Region. Additionally, a Sub-Region may have its own Sub-Regions and configuration. A Sub-Region does not inherit attributes from its parent Region. Regions types may be mixed and matched subject to Pivotal GemFire constraints. A Sub-Region is naturally declared as a child element of a Region. The Sub-Region's `name` attribute is the simple name. The preceding example might be configured as follows:

```
<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 Sub-Region. Sub-Regions are created with bean names (/Customer/Address and /Employee/Address, respectively, in this case). So they may be injected into other application beans, such as a `GemfireTemplate` , that need them by using the full path name of the Region. The full pathname of the Region should also be used in OQL query strings.

5.5.7. Region Templates

Spring Data for Pivotal GemFire also supports Region templates.

This feature allows developers to define common Region configuration and attributes once and reuse the configuration among many Region bean definitions declared in the Spring `ApplicationContext` .

Spring Data for Pivotal GemFire includes five Region template tags in its namespace:

Table 2. Region Template Tags

Tag Name	Description

Tag Name	Description
`<gfe:region-template>`	Defines common generic Region attributes. Extends `regionType` in the XML namespace.
`<gfe:local-region-template>`	Defines common 'Local' Region attributes. Extends `localRegionType` in the XML namespace.
`<gfe:partitioned-region-template>`	Defines common 'PARTITION' Region attributes. Extends `partitionedRegionType` in the XML namespace.
`<gfe:replicated-region-template>`	Defines common 'REPLICATE' Region attributes. Extends `replicatedRegionType` in the XML namespace.
`<gfe:client-region-template>`	Defines common 'Client' Region attributes. Extends `clientRegionType` in the XML 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 inherits its configuration. Region templates may even inherit from other Region templates.

The following example shows one 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 Sub-Regions as well. Notice that 'TemplateBasedPartitionRegion' extends 'PartitionRegionTemplate', which extends 'ExtendedRegionTemplate', which extends 'BaseRegionTemplate'. Attributes and sub-elements defined in subsequent, inherited Region bean definitions override what is in the parent.

How Templating Works

Spring Data for Pivotal GemFire applies Region templates when the Spring `ApplicationContext` configuration metadata is parsed, and therefore, Region templates must be declared in the order of inheritance. In other words, parent templates must be defined before child templates. Doing so ensures that the proper configuration is applied, especially when element attributes or sub-elements are overridden.

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

Region templates are single-inheritance.

Caution concerning Regions, Sub-Regions and Lookups

Previously, one of the underlying properties of the `replicated-region`, `partitioned-region`, `local-region`, and `client-region` elements in the Spring Data for Pivotal GemFire 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 Pivotal GemFire 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 for Pivotal GemFire `<gfe:*-region>` XML namespace elements now include a `ignore-if-exists` attribute, which reinstates the old behavior by first performing a lookup of an existing Region identified by name before attempting to create the Region. If an existing Region is found by name and `ignore-if-exists` is set to `true`, then the Region bean definition defined in Spring configuration is ignored.

The Spring team highly recommends that the `replicated-region`, `partitioned-region`, `local-region`, and `client-region` XML namespace elements be strictly used for defining new Regions only. One problem that could arise when the Regions defined by these elements already exist and the Region elements perform a lookup first is, if you defined different Region semantics and behaviors for eviction, expiration, subscription, and so on in your application config, then the Region definition might not match and could exhibit contrary behaviors to those required by the application. Even worse, you might want to define the Region as a distributed Region (for example, `PARTITION`) when, in fact, the existing Region definition is local only.

Recommended Practice - Use only `replicated-region`, `partitioned-region`, `local-region`, and `client-`

> `region` XML namespace elements to define new Regions.

Consider the following native Pivotal GemFire `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>
```

Further, 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 inject a reference to the `Customers/Accounts` Region in our application DAO. Consequently, it is not uncommon for a developer to define beans for some or all of these Regions in Spring XML configuration metadata as follows:

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

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

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

</beans>
```

The `Customers/Accounts` and `Customers/Accounts/Orders` Regions are referenced as beans in the Spring container as `Customers/Accounts` and `Customers/Accounts/Orders`, respectively. The nice thing about using the `lookup-region` element and the corresponding syntax (described earlier) is that it lets you reference a Sub-Region directly without unnecessarily defining a bean for the parent Region (`Customers`, in this case).

Consider the following bad example, which changes the configuration metadata syntax to use the nested format:

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

Now consider another bad example which uses the top-level `replicated-region` element along with the `ignore-if-exists` attribute set to perform a lookup first:

```
<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>
```

The Region beans defined in the Spring `ApplicationContext` consist of the following: `{ "Customers", "/Customers/Accounts", "/Customers/Accounts/Orders" }`. This means the dependency injected reference shown in the earlier example (that is, `@Resource(name = "Customers/Accounts")`) is now broken, since no bean with name `Customers/Accounts` is actually defined. For this reason, you should not configure Regions as shown in the two preceding examples.

Pivotal GemFire is flexible in referencing both parent Regions and Sub-Regions 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 `Customers/Accounts` . However, Spring Data for Pivotal GemFire is very specific when it comes to naming beans after Regions. It always uses the forward slash (/) to represent Sub-Regions (for example, `/Customers/Accounts`).

Therefore, you should use the non-nested `lookup-region` syntax shown earlier or define direct references with a leading forward slash (/), as follows:

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

The earlier example, where the nested `replicated-region` elements were used to reference the Sub-Regions, shows the problem stated earlier. Are the Customers, Accounts and Orders Regions and Sub-Regions persistent or not? They are not persistent, because the Regions were defined in the native Pivotal GemFire `cache.xml` configuration file as `REPLICATE` and exist before the cache bean is initialized (once the `<gfe:cache>` element is processed).

5.5.8. Data Eviction (with Overflow)

Based on various constraints, each Region can have an eviction policy in place for evicting data from memory. Currently, in Pivotal GemFire, 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 for Pivotal GemFire supports all eviction policies (entry count, memory, and heap usage) for PARTITION Regions, REPLICATE Regions, and client, local Regions by 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, you can specify the following configuration:

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

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

When configuring Regions for overflow, you should configure the storage through the `disk-store` element for maximum efficiency.

For a detailed description of eviction policies, see the Pivotal GemFire documentation on Eviction.

5.5.9. Data Expiration

Pivotal GemFire lets you 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 or memory usage. Once an entry expires, it may no longer be accessed from the cache.

Pivotal GemFire 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 to entries in the Region. Spring Data for Pivotal GemFire provides `<region-ttl>`, `<region-tti>`, `<entry-ttl>`, and `<entry-tti>` Region child elements to specify timeout values and expiration actions.

The following example shows a `PARTITION` Region with expiration values set:

```
<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, see the Pivotal GemFire documentation on expiration.

Annotation-based Data Expiration

With Spring Data for Pivotal GemFire, you can define expiration policies and settings on individual Region entry values (or, to put it differently, directly on application domain objects). For instance, you can define expiration policies on a Session-based application domain object as follows:

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

You can also specify expiration type specific settings on Region entries by using the `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` annotations for Idle Timeout (TTI) and Time-to-Live (TTL) expiration, respectively, as the following example shows:

```
@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 in the preceding example. Neither `@IdleTimeoutExpiration` nor `@TimeToLiveExpiration` overrides the other. Rather, they compliment each other when different Region entry expiration policies, such as TTL and TTI, are configured.

All `@Expiration`-based annotations apply only to Region entry values. Expiration for a Region is not covered by Spring Data for Pivotal GemFire's expiration annotation support. However, Pivotal GemFire and Spring Data for Pivotal GemFire do let you set Region expiration by using the SDG XML namespace, as follows:

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

Spring Data for Pivotal GemFire's `@Expiration` annotation support is implemented with Pivotal GemFire's `CustomExpiry` interface. See Pivotal GemFire's documentation on configuring data expiration for more details

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

To use Spring Data for Pivotal GemFire to configure specific Pivotal GemFire Regions to appropriately apply the expiration policy to your application domain objects annotated with `@Expiration`-based annotations, you must:

1. Define a bean in the Spring `ApplicationContext` of type `AnnotationBasedExpiration` by using the appropriate constructor or one of the convenient factory methods. When configuring expiration for a specific expiration type, such as Idle Timeout (TTI) or Time-to-Live (TTL), you should use one of the factory methods in the `AnnotationBasedExpiration` class, as follows:

```
<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>
```

To configure Idle Timeout (TTI) Expiration instead, 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 are stored in the Region with expiration policies and custom settings by using one of Spring Data for Pivotal GemFire's `@Expiration` annotations: `@Expiration`, `@IdleTimeoutExpiration`, or `@TimeToLiveExpiration`
3. (optional) In cases where particular application domain objects have not been annotated with Spring Data for Pivotal GemFire's `@Expiration` annotations at all, but the Pivotal GemFire Region is configured to use SDG's custom `AnnotationBasedExpiration` class to determine the expiration policy and settings for objects stored in the Region, you can 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 for Pivotal GemFire's `@Expiration` annotations use a `String` as the attribute type rather than, and perhaps more appropriately, being strongly typed — for example, `int` for 'timeout' and SDG's `ExpirationActionType` for 'action'. Why is that?

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

For instance, a developer can specify both the expiration 'timeout' and 'action' by using property placeholders in the `@Expiration` annotation attributes, as the following example shows:

```
@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 can 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 convenient both when multiple application domain objects might share similar expiration policies and when you wish to externalize the configuration.

However, you may want more dynamic expiration configuration determined by the state of the running system. This is where the power of SpEL comes into play and is the recommended approach, actually. Not only can you refer to beans in the Spring container and access bean properties, invoke methods, and so on, but the values for expiration 'timeout' and 'action' can be strongly typed. Consider the following example (which builds on the preceding example):

```
<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, you can define a timeout and an action as follows:

```
@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 than a simple instance of `java.util.Properties`. In the preceding example, the `properties` element (`expirationSettings`) uses SpEL to base the action value on the actual `ExpirationAction` enumerated type, quickly leading to identified failures if the enumerated type ever changes.

As an example, all of this has been demonstrated and tested in the Spring Data for Pivotal GemFire test suite. See the source for further details.

5.5.10. Data Persistence

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

To enable persistence with Spring Data for Pivotal GemFire, set the `persistent` attribute to `true` on any of the `<*-region>` elements, as the following example shows:

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

Persistence may also be configured by setting the `data-policy` attribute. To do so, set the attribute's value to one of Pivotal GemFire's DataPolicy settings, as the following example shows:

```
<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 it is also explicitly set. If the `persistent` attribute is set to `false` but a persistent `DataPolicy` was specified (such as `PERSISTENT_REPLICATE` or `PERSISTENT_PARTITION`), an initialization exception is thrown.

For maximum efficiency when persisting Regions, you should configure the storage through the `disk-store` element. The `DiskStore` is referenced by using the `disk-store-ref` attribute. Additionally, the Region may perform disk writes synchronously or asynchronously. The following example shows a synchronous `DiskStore`:

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

This is discussed further in Configuring a DiskStore.

5.5.11. Subscription Policy

Pivotal GemFire allows configuration of peer-to-peer (P2P) event messaging to control the entry events that the Region receives. Spring Data for Pivotal GemFire provides the `<gfe:subscription/>` sub-element to set the subscription policy on `REPLICATE` and `PARTITION` Regions to either `ALL` or `CACHE_CONTENT`. The following example shows a region with its subscription policy set to `CACHE_CONTENT`:

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

5.5.12. Local Region

Spring Data for Pivotal GemFire offers a dedicated `local-region` element for creating local Regions. Local Regions, as the name implies, are standalone, meaning that they do not share data with any other distributed system member. Other than that, all common Region configuration options apply.

The following example shows a minimal declaration (again, the example relies on the Spring Data for Pivotal GemFire XML namespace naming conventions to wire the cache):

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

In the preceding example, a local Region is created (if a Region by the same name does not already exist). The name of the Region is the same as the bean ID (`exampleLocalRegion`), and the bean assumes the existence of a Pivotal GemFire cache named `gemfireCache`.

5.5.13. 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 replicas.

All common configuration options are available for REPLICATE Regions. Spring Data for Pivotal GemFire offers a `replicated-region` element. The following example shows a minimal declaration:

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

See Pivotal GemFire’s documentation on Distributed and Replicated Regions for more details.

5.5.14. Partitioned Region

The Spring Data for Pivotal GemFire XML namespace also supports `PARTITION` Regions.

To quote the Pivotal GemFire docs:

“A partitioned region is a region where data is divided between peer servers hosting the region so that each peer stores a subset of the data. When using a partitioned region, applications are presented with a logical view of the region that looks like a single map containing all of the data in the region. Reads or writes to this map are transparently routed to the peer that hosts the entry that is the target of the operation. Pivotal GemFire divides the domain of hashcodes into buckets.

Each bucket is assigned to a specific peer, but may be relocated at any time to another peer in order to improve the utilization of resources across the cluster."

A `PARTITION` Region is created by using the `partitioned-region` element. Its configuration options are similar to that of the `replicated-region` with the addition of partition-specific features, such as the number of redundant copies, total maximum memory, number of buckets, partition resolver, and so on.

The following example shows how to set up a `PARTITION` Region with two 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>
```

See Pivotal GemFire'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 options are in addition to the common Region configuration options described earlier.

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.

Name	Values	Description
recovery-delay	any long value	The delay in milliseconds that existing members wait before satisfying redundancy after another member crashes. -1 (the default) indicates that redundancy is not recovered after a failure.
startup-recovery-delay	any long value	The delay in milliseconds that new members wait before satisfying redundancy. -1 indicates that adding new members does not trigger redundancy recovery. The default is to recover redundancy immediately when a new member is added.

5.5.15. Client Region

Pivotal GemFire supports various deployment topologies for managing and distributing data. The topic of Pivotal GemFire topologies is beyond the scope of this documentation. However, to quickly recap, Pivotal GemFire's supported topologies can be classified as: peer-to-peer (p2p), client-server, and wide area network (WAN). In the last two configurations, it is common to declare client Regions that connect to a cache server.

Spring Data for Pivotal GemFire offers dedicated support for each configuration through its client-cache elements: `client-region` and `pool`. As the names imply, `client-region` defines a client Region, while `pool` defines a Pool of connections used and shared by the various client Regions.

The following example shows 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` instances as well as a `CacheLoader` and a `CacheWriter`. It also requires a connection `Pool` for connecting to a set of either Locators or servers. Each client Region can have its own `Pool`, or they can share the same one. If a Pool is not specified, then the "DEFAULT" Pool will be used.

In the preceding example, the `Pool` is configured with a Locator. A Locator is a separate process used to discover cache servers and peer data members in the distributed system and is recommended for production systems. It is also possible to configure the `Pool` to connect directly to one or more cache servers by using the `server` element.

For a full list of options to set on the client and especially on the `Pool`, see the Spring Data for Pivotal GemFire schema ("Spring Data for Pivotal GemFire Schema") and Pivotal GemFire's documentation on Client-Server Configuration.

Client Interests

To minimize network traffic, each client can separately define its own 'interests' policies, indicating to Pivotal GemFire the data it actually requires. In Spring Data for Pivotal GemFire, 'interests' can be defined for each client Region separately. Both key-based and regular expression-based interest types are supported.

The following example shows both key-based and regular expression-based `interest` types:

```
<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 by using the regular expression, `".\*"`.

The `<gfe:*-interest>` key and regular expression elements support three 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 one 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 servers for the client is maintained while the client is disconnected. When the client reconnects, the client receives any events that occurred while the client was disconnected from the servers in the cluster.

A subscription queue on the servers in the cluster is maintained for each `Pool` of connections defined in the client where a 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 (that is, connections), potentially up to a specified timeout. If the client does not return within a given time frame the client Pool subscription queue is destroyed in order to reduce resource consumption on servers in the cluster. If the subscription queue is not `durable`, it is destroyed immediately when the client disconnects. You need to decide whether your client should receive events that came while it was disconnected or if it needs to receive only 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 interest registration does not do much good without enabling subscription on the `Pool`, as mentioned earlier. In fact, it is an error to attempt interest registration without subscription enabled. The following example shows how to do so:

```
<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 one, and the "primary" subscription queue (that is, the server) goes down, then a "secondary" subscription queue takes 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 are 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, when the application keeps a time series of events that occurred, conflation is going to hinder that use case. The default value is `false`. The following example shows a Region configuration on the server, for which the client contains a corresponding client `[CACHING_]PROXY` Region with interests in keys in this server Region:

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

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

```
<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.

See Pivotal GemFire's documentation on Client-to-Server Event Distribution for more details.

5.5.16. JSON Support

Pivotal GemFire has support for caching JSON documents in Regions, along with the ability to query stored JSON documents using the Pivotal GemFire OQL (Object Query Language). JSON documents are stored internally as PdxInstance types by using the JSONFormatter class to perform conversion to and from JSON documents (as a `String`).

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

In addition, Java objects written to JSON configured Regions are automatically converted to JSON using Jackson's `ObjectMapper`. When these values are read back, they are 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 in 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 are 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.

> Certain Region operations (specifically those that use Pivotal GemFire's proprietary `Region.Entry`, such

as: `entries(boolean)`, `entrySet(boolean)` and `getEntry()` type) are not targeted for AOP advice. In addition, the `entrySet()` method (which returns a `Set<java.util.Map.Entry<?, ?>>`) is also not affected.

The following example configuration shows how to set the `pretty-print` and `convert-returned-collections` attributes:

```
<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

Pivotal GemFire allows indexes (also sometimes pluralized as indices) to be created on Region data to improve the performance of OQL (Object Query Language) queries.

In Spring Data for Pivotal GemFire, indexes are declared with the `index` element, as the following example shows:

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

In Spring Data for Pivotal GemFire's XML schema (also called the SDG XML namespace), `index` bean declarations are not bound to a Region, unlike Pivotal GemFire's native `cache.xml`. Rather, they are top-level elements similar to `<gfe:cache>` element. This lets you declare any number of indexes on any Region, whether they were just created or already exist — a significant improvement over Pivotal GemFire's native `cache.xml` format.

An `Index` must have a name. You can give the `Index` an explicit name by using the `name` attribute. Otherwise, the bean name (that is, the 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 (that is, the Region identified in the `from` clause) along with what criteria (that is, `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 look up the objects stored in the Region.

Consider the following example, which has a `lastName` property:

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

  @Id
  Long id;

  String lastName;
  String firstName;

  ...
}
```

Now consider the following example, which has an application-defined SDG Repository to query for `Customer` objects:

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

  Customer findByLastName(String lastName);
```

```
  ...
}
```

The SDG Repository finder/query method results in the following OQL statement being generated and ran:

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

Therefore, you might want to create an `Index` with a statement similar to the following:

```
<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 specific to Spring Data for Pivotal GemFire. It is a feature of Pivotal GemFire.

The `Index type` may be one of three enumerated values defined by Spring Data for Pivotal GemFire's `IndexType` enumeration: `FUNCTIONAL`, `HASH`, and `PRIMARY_KEY`.

Each of the enumerated values corresponds to one of the `QueryService create[|Key|Hash]Index` methods invoked when the actual `Index` is to be created (or "defined" — you can find more on "defining" indexes in the next section). 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 for Pivotal GemFire XML schema for a full set of options.

For more information on indexing in Pivotal GemFire, see "Working with Indexes" in Pivotal GemFire's User Guide.

5.6.1. Defining Indexes

In addition to creating indexes up front as `Index` bean definitions are processed by Spring Data for Pivotal GemFire on Spring container initialization, you may also define all of your application indexes prior to creating them by using the `define` attribute, as follows:

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

When `define` is set to `true` (it defaults to `false`), it does not actually create the `Index` at that moment. All "defined" Indexes are created all at once, when the Spring `ApplicationContext` is "refreshed" or, to put it differently, when a `ContextRefreshedEvent` is published by the Spring container. Spring Data for Pivotal GemFire registers itself as an `ApplicationListener` listening for the `ContextRefreshedEvent`. When fired, Spring Data for Pivotal GemFire calls `QueryService.createDefinedIndexes()`.

Defining indexes and creating them all at once boosts speed and efficiency when creating indexes.

See "Creating Multiple Indexes at Once" for more details.

5.6.2. `IgnoreIfExists` and `Override`

Two Spring Data for Pivotal GemFire `Index` configuration options warrant special mention: `ignoreIfExists` and `override`.

These options correspond to the `ignore-if-exists` and `override` attributes on the `<gfe:index>` element in Spring Data for Pivotal GemFire's XML namespace, respectively.

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

These options are only available in Spring Data for Pivotal GemFire and exist to workaround known limitations with Pivotal GemFire. Pivotal GemFire has no equivalent options or functionality.

Each option significantly differs in behavior and entirely depends on the type of Pivotal GemFire `Index` exception thrown. This also means that neither option has any effect if a Pivotal GemFire Index-type exception is not thrown. These options are meant to specifically handle Pivotal GemFire `IndexExistsException` and `IndexNameConflictException` instances, which can occur for various, sometimes obscure reasons. The exceptions have the following causes:

- An `IndexExistsException` is thrown when there exists another `Index` with the same definition but a 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 for Pivotal GemFire's default behavior is to fail-fast, always. So, neither `Index` *Exception* are "handled" by default. These `Index` exceptions are wrapped in a SDG `GemfireIndexException` and rethrown. If you wish for Spring Data for Pivotal GemFire to handle them for you, you can set either of these `Index` bean definition options to `true`.

`IgnoreIfExists` always takes precedence over `Override`, primarily because it uses fewer resources, simply because 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 or declaration is simply ignored, and the existing `Index` is returned.

There is little consequence in returning the existing `Index`, since the `index` bean definition is the same, as determined by Pivotal GemFire itself, not SDG.

However, this also means that no `Index` with the "name" specified in your `index` bean definition or declaration actually exists from Pivotal GemFire's perspective (that is, with `QueryService.getIndexes()`). Therefore, you should be careful when writing OQL query statements that use query hints, especially query hints that refer to the application `Index` being ignored. Those query hints need to be changed.

When an `IndexNameConflictException` is thrown and `ignoreIfExists` is set to `true` (or `<gfe:index ignore-if-exists="true">`), the `Index` that would have been created by this `index` bean definition or declaration is also ignored, and the "existing" `Index` is again returned, as 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. For a `IndexNameConflictException`, while the names of the conflicting indexes are the same, the definitions could be different. This situation 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. Consequently, you should verify your `Index` names.

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 by using the Pivotal GemFire API (the only means available).

`Override` Behavior

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

Spring Data for Pivotal GemFire can only accomplish this by using Pivotal GemFire'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 that refer to the old `Index` by name must be changed.

When an `IndexNameConflictException` is thrown and `override` is set to `true` (or `<gfe:index override="true">`), the existing `Index` can potentially be re-defined. We 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 returns the existing `Index` as is, even on `override`. There is no harm in this behavior, 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 Pivotal GemFire's APIs. If it cannot be found, nothing happens and a SDG `GemfireIndexException` is thrown that wraps the `IndexNameConflictException`.

However, when the definition of the existing `Index` is different, SDG attempts to re-create the `Index` by using the `Index` definition specified in the `index` bean definition. Make sure this is what you want and make sure the `index` bean 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 Pivotal GemFire (Spring Data for Pivotal GemFire, Pivotal GemFire Cluster Config, Pivotal GemFire native `cache.xml`, the API, and so on). You should definitely prefer one configuration method 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 (for example, `X`), Pivotal GemFire distributes the `Index` definition (and name) to other peer members in the cluster that also host the same `PARTITION` Region (that is, "X"). The distribution of this `Index` definition to, and subsequent creation of, this `Index` by peer members is on a need-to-know basis (that is, by peer member hosting the same PR) is performed asynchronously.

During this window of time, it is possible that these pending PR `Indexes` cannot be identified by Pivotal GemFire — such as with a call to `QueryService.getIndexes()` with `QueryService.getIndexes(:Region)`, or even with `QueryService.getIndex(:Region, indexName:String)`.

As a result, the only way for SDG or other Pivotal GemFire cache client applications (not involving Spring) to know for sure is to attempt to create the `Index`. If it fails with either an `IndexNameConflictException` or even an `IndexExistsException`, the application knows there is a problem. This is because the `QueryService` `Index` creation waits on pending `Index` definitions, whereas the other Pivotal GemFire API calls do not.

In any case, SDG makes a best effort and attempts to inform you what has happened or is happening and tell you the corrective action. Given that all Pivotal GemFire `QueryService.createIndex(..)` methods are synchronous, blocking operations, the state of Pivotal GemFire should be consistent and accessible after either of these index-type exceptions are thrown. Consequently, SDG can inspect the state of the system and act accordingly, based on your configuration.

In all other cases, SDG embraces a fail-fast strategy.

5.7. Configuring a DiskStore

Spring Data for Pivotal GemFire supports `DiskStore` configuration and creation through the `disk-store` element, as the following example shows:

```
<gfe:disk-store id="Example" auto-compact="true" max-oplog-size="10"
                queue-size="50" time-interval="9999">
    <gfe:disk-dir location="/disk/location/one" max-size="20"/>
    <gfe:disk-dir location="/disk/location/two" max-size="20"/>
</gfe:disk-store>
```

`DiskStore` instances are used by Regions for file system persistent backup and overflow of evicted entries as well as persistent backup for WAN Gateways. Multiple Pivotal GemFire components may share the same `DiskStore`. Additionally, multiple file system directories may be defined for a single `DiskStore`, as shown in the preceding example.

See Pivotal GemFire's documentation for a complete explanation of Persistence and Overflow and configuration options on `DiskStore` instances.

5.8. Configuring the Snapshot Service

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

As the Pivotal GemFire documentation explains, snapshots let you 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 combine Spring Data for Pivotal GemFire's Snapshot Service support with Spring's bean definition profiles to load snapshot data specific to the environment as necessary.

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

For example, you can define cache-wide snapshots to be loaded as well as saved by using a couple of 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 exports as you like. You can define only imports or only exports. The file locations and directory paths can be absolute or relative to the Spring Data for Pivotal GemFire application, which is the JVM process's working directory.

The preceding example is pretty simple, and the Snapshot Service defined in this case refers to the Pivotal GemFire cache instance with the default name of `gemfireCache` (as described in Configuring a Cache). If you name your cache bean definition something other than the default, you can use the `cache-ref` attribute to refer to the cache bean by name, as follows:

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

You can also define a Snapshot Service for a particular Region by specifying the `region-ref` attribute, as follows:

```
<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 for Pivotal GemFire's `SnapshotServiceFactoryBean` resolves the `region-ref` attribute value to a Region bean defined in the Spring container and creates a `RegionSnapshotService`. The snapshot import and export definitions function the same way. However, the `location` must refer to a file on an export.

Pivotal GemFire is strict about imported snapshot files actually existing before they are referenced. For exports, Pivotal GemFire creates the snapshot file. If the snapshot file for export already exists, the data is overwritten.

Spring Data for Pivotal GemFire 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. Doing so is useful, for example, when data exported from one Region is used to feed the import of another Region.

5.8.1. Snapshot Location

With the cache-based Snapshot Service (that is, a `CacheSnapshotService`) you 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.

Of course, you can use the overloaded `load(:File[], :SnapshotFormat, :SnapshotOptions)` method to get specific about which snapshot files to load into the Pivotal GemFire cache.

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

Therefore, Spring Data for Pivotal GemFire lets you 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 for Pivotal GemFire conveniently extracts the provided zip file and treats it as 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 Pivotal GemFire's `SnapshotFilter` interface and are used to filter Region entries for inclusion into the Region on import and for inclusion into the snapshot on export.

Spring Data for Pivotal GemFire lets you use snapshot filters on import and export by using the `filter-ref` attribute or an anonymous, nested bean definition, as the following example shows:

```
<gfe:cache/>

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

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

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

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

In addition, you can express more complex snapshot filters by using the `ComposableSnapshotFilter` class. This class implements Pivotal GemFire's SnapshotFilter interface as well as the Composite software design pattern.

In a nutshell, the Composite software design pattern lets you compose multiple objects of the same type and treat the aggregate as single instance of the object type — a powerful and useful abstraction.

`ComposableSnapshotFilter` has two factory methods, `and` and `or`. They let you logically combine individual snapshot filters using the AND and OR logical operators, respectively. The factory methods take a list of `SnapshotFilters`.

The following example shows a definition for a `ComposableSnapshotFilter`:

```
<bean id="activeUsersSinceFilter" class="org.springframework.data.gemfire.snapshot.filter.ComposableSnapshotFilter"
      factory-method="and">
  <constructor-arg index="0">
    <list>
```

```
      <bean class="org.example.app.gemfire.snapshot.filter.ActiveUsersFilter"/>
      <bean class="org.example.app.gemfire.snapshot.filter.UsersSinceFilter"
            p:since="2015-01-01"/>
    </list>
  </constructor-arg>
</bean>
```

You could then go on to combine the `activesUsersSinceFilter` with another filter by using `or`, as follows:

```
<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.gemfire.snapshot.filter.CovertUsersFilter"/>
    </list>
  </constructor-arg>
</bean>
```

5.8.3. Snapshot Events

By default, Spring Data for Pivotal GemFire uses Pivotal GemFire's Snapshot Services on startup to import data and on 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 for Pivotal GemFire defines two additional Spring application events, extending Spring's `ApplicationEvent` class for imports and exports, respectively: `ImportSnapshotApplicationEvent` and `ExportSnapshotApplicationEvent`.

The two application events can be targeted for the entire Pivotal GemFire cache or for individual Pivotal GemFire Regions. The constructors in these classes accept an optional Region pathname (such as `/Example`) as well as zero or more `SnapshotMetadata` instances.

The array of `SnapshotMetadata` overrides the snapshot metadata defined by `<gfe-data:snapshot-import>` and `<gfe-data:snapshot-export>` sub-elements, which are used in cases where snapshot application events do not explicitly provide `SnapshotMetadata`. Each individual `SnapshotMetadata` instance can define its own `location` and `filters` properties.

All snapshot service beans defined in the Spring `ApplicationContext` receive import and export snapshot application events. However, only matching Snapshot Service beans process import and export events.

A Region-based `[Import|Export]SnapshotApplicationEvent` matches if the Snapshot Service bean defined is a `RegionSnapshotService` and its Region reference (as determined by the `region-ref` attribute) matches the Region's pathname, as specified by the snapshot application event.

A Cache-based `[Import|Export]SnapshotApplicationEvent` (that is, 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.

You can use Spring's `ApplicationEventPublisher` interface to fire import and export snapshot application events from your application as follows:

```
@Component
public class ExampleApplicationComponent {

  @Autowired
  private ApplicationEventPublisher eventPublisher;
```

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

  public void someMethod() {

    ...

    File dataSnapshot = new File(System.getProperty("user.dir"), "/path/to/export/data.snapshot");

    SnapshotFilter myFilter = ...;

    SnapshotMetadata exportSnapshotMetadata =
        new SnapshotMetadata(dataSnapshot, myFilter, null);

    ExportSnapshotApplicationEvent exportSnapshotEvent =
        new ExportSnapshotApplicationEvent(this, example.getFullPath(), exportSnapshotMetadata)

    eventPublisher.publishEvent(exportSnapshotEvent);

    ...
  }
}
```

In the preceding example, only the `/Example` Region's Snapshot Service bean picks up and handles the export event, saving the filtered, "/Example" Region's data to the `data.snapshot` file in a sub-directory of the application's working directory.

Using the Spring application events and messaging subsystem is a good way to keep your application loosely coupled. You can also use Spring's Scheduling services to fire snapshot application events on a periodic basis.

5.9. Configuring the Function Service

Spring Data for Pivotal GemFire provides annotation support for implementing, registering and executing Pivotal GemFire Functions.

Spring Data for Pivotal GemFire also provides XML namespace support for registering Pivotal GemFire Functions for remote function execution.

See Pivotal GemFire's documentation for more information on the Function execution framework.

Pivotal GemFire 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, as the following example shows:

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

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

6. Configuring WAN Gateways

WAN Gateways provides a way to synchronize Pivotal GemFire Distributed Systems across geographic locations. Spring Data for Pivotal GemFire provides XML namespace support for configuring WAN Gateways as illustrated in the following

examples.

6.1. WAN Configuration in Pivotal GemFire 7.0

In the following example, `GatewaySenders` are configured for a `PARTITION` Region by adding child elements (`gateway-sender` and `gateway-sender-ref`) to the Region. A `GatewaySender` may register `EventFilters` and `TransportFilters`.

The following example also shows a sample configuration of an `AsyncEventQueue`, which must also be auto-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`, as follows:

```
<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>
```

See the Pivotal GemFire documentation for a detailed explanation of all the configuration options.

7. Bootstrapping Pivotal GemFire with the Spring Container using Annotations

Spring Data for Pivotal GemFire (SDG) 2.0 introduces a new annotation-based configuration model to configure and bootstrap Pivotal GemFire using the Spring container.

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

Let's get started!

If you would like to get started even faster, refer to the Quick Start section.

7.1. Introduction

Pivotal GemFire can be difficult to setup and use correctly, given all the configuration properties and different configuration options:

- Java API
- `cache.xml`
- *Gfsh* with Cluster Configuration
- Spring XML/Java-based configuration

Further complexity arises from the different supported topologies:

- (client/server
- P2P
- WAN)
- distributed system design patterns (such as shared-nothing architecture).

The annotation-based configuration model aims to simplify all this and more.

The annotation-based configuration model is an alternative to XML-based configuration using Spring Data for Pivotal GemFire's XML namespace. With XML, you could use the `gfe` XML schema for configuration and the `gfe-data` XML schema for data access. See "Bootstrapping Pivotal GemFire with the Spring Container" for more details.

As of SDG 2.0, the annotation-based configuration model does not yet support the configuration of Pivotal GemFire's WAN components and topology.

Like Spring Boot, Spring Data for Pivotal GemFire's annotation-based configuration model was designed as an opinionated, convention-over-configuration approach for using Pivotal GemFire. Indeed, this annotation-based configuration model was inspired by Spring Boot as well as several other Spring and Spring Data projects, collectively.

By following convention, all annotations provide reasonable and sensible defaults for all configuration attributes. The default value for a given annotation attribute directly corresponds to the default value provided in Pivotal GemFire for the same configuration property.

The intention is to let you enable Pivotal GemFire features or an embedded services by declaring the appropriate annotation on your Spring `@Configuration` or `@SpringBootApplication` class without needing to unnecessarily configure a large number of properties just to use the feature or service.

Again, *getting started*, *quickly* and as *easily*, is the primary objective.

However, the option to customize the configuration metadata and behavior of Pivotal GemFire is there if you need it, and Spring Data for Pivotal GemFire's annotation-based configuration quietly backs away. You need only specify the configuration attributes you wish to adjust. Also, as we will see later in this document, there are several ways to configure a Pivotal GemFire feature or embedded service by using the annotations.

You can find all the new SDG Java `Annotations` in the `org.springframework.data.gemfire.config.annotation` package.

7.2. Configuring Pivotal GemFire Applications with Spring

Like all Spring Boot applications that begin by annotating the application class with `@SpringBootApplication`, a Spring Boot application can easily become a Pivotal GemFire cache application by declaring any one of three main annotations:

- `@ClientCacheApplication`
- `@PeerCacheApplication`
- `@CacheServerApplication`

These three annotations are the Spring application developer's starting point when working with Pivotal GemFire.

To realize the intent behind these annotations, you must understand that there are two types of cache instances that can be created with Pivotal GemFire: a client cache or a peer cache.

You can configure a Spring Boot application as a Pivotal GemFire cache client with an instance of `ClientCache`, which can communicate with an existing cluster of Pivotal GemFire servers used to manage the application's data. The client-server topology is the most common system architecture employed when using Pivotal GemFire and you can make your Spring Boot application a cache client, with a `ClientCache` instance, simply by annotating it with `@ClientCacheApplication`.

Alternatively, a Spring Boot application may be a peer member of a Pivotal GemFire cluster. That is, the application itself is just another server in a cluster of servers that manages data. The Spring Boot application creates an "embedded", peer `Cache` instance when you annotate your application class with `@PeerCacheApplication`.

By extension, a peer cache application may also serve as a `CacheServer` too, allowing cache 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 creates a peer `Cache` instance along with the `CacheServer` that allows cache clients to connect.

A Pivotal GemFire server is not necessarily a cache server by default. That is, a server is not necessarily set up to serve cache clients just because it is a server. A Pivotal GemFire server can be a peer member (data node) of the cluster managing data without serving any clients while other peer members in the cluster are indeed set up to serve clients in addition to managing data. It is also possible to set up certain peer members in the cluster as non-data nodes, called data accessors, which do not store data, but act as a proxy to service clients as `CacheServers`. Many different topologies and cluster arrangements are supported by Pivotal GemFire, but are beyond the scope of this document.

By way of example, if you want to create a Spring Boot cache client application, start with the following:

Spring-based Pivotal GemFire `ClientCache` *application*

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

Or, if you want to create a Spring Boot application with an embedded peer `Cache` instance, where your application will be a server and peer member of a cluster (distributed system) formed by Pivotal GemFire, start with the following:

Spring-based Pivotal GemFire embedded peer `Cache` *application*

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

Alternatively, you can use the `@CacheServerApplication` annotation in place of `@PeerCacheApplication` to create both an embedded peer `Cache` instance along with a `CacheServer` running on `localhost`, listening on the default cache server port, `40404`, as follows:

Spring-based Pivotal GemFire embedded peer `Cache` *application with* `CacheServer`

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

7.3. Client/Server Applications In-Detail

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

A cache client can connect to one or more Locators in the Pivotal GemFire cluster instead of directly to a `CacheServer`. The advantage of using Locators over direct `CacheServer` connections is that Locators provide metadata about the cluster to which the client is connected. This metadata includes information such as which servers contain the data of interest or which servers have the least amount of load. A client `Pool` in conjunction with a Locator also provides fail-over capabilities in case a `CacheServer` crashes. By enabling the `PARTITION` Region (PR) single-hop feature in the client `Pool`, the client is routed directly to the server containing the data requested and needed by the client.

Locators are also peer members in a cluster. Locators actually constitute what makes up a cluster of Pivotal GemFire nodes. That is, all nodes connected by a Locator are peers in the cluster, and new members use Locators to join a cluster and find other members.

By default, Pivotal GemFire sets up a "DEFAULT" `Pool` connected to a `CacheServer` running on `localhost`, listening on port `40404` when a `ClientCache` instance is created. A `CacheServer` listens on port `40404`, accepting connections on all system NICs. You do not need to do anything special to use 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 ready to go.

If you prefer, you can even start your servers with Gfsh's `start server` command. Your Spring Boot `@ClientCacheApplication` can still connect to the server regardless of how it was started. However, you may prefer to configure and start your servers by using the Spring Data for Pivotal GemFire approach since a properly annotated Spring Boot application class is far more intuitive and easier to debug.

As an application developer, you will no doubt want to customize the "DEFAULT" `Pool` set up by Pivotal GemFire to possibly connect to one or more Locators, as the following example demonstrates:

Spring-based Pivotal GemFire `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 as well. The `servers` attribute can be used to specify one or more nested `@Server` annotations that let the cache client connect directly to one or more servers, if necessary.

You can use either the `locators` or `servers` attribute, but not both (this is enforced by Pivotal GemFire).

You can also configure additional `Pool` instances (other than the "DEFAULT" `Pool` provided by Pivotal GemFire when a `ClientCache` instance is created with the `@ClientCacheApplication` annotation) by using the `@EnablePool` and `@EnablePools` annotations.

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

The following example uses the `@EnablePool` and `@EnablePools` annotations:

Spring-based Pivotal GemFire `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 later, the value of the `name` attribute corresponds to both the name of the `Pool` bean created in the Spring container as well as the name used to reference the corresponding configuration properties. It is also the name of the `Pool` registered and used by Pivotal GemFire.

Similarly, on the server, you can configure multiple `CacheServers` that a client can connect to, as follows:

Spring-based Pivotal GemFire `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 { .. }
```

Like `@EnablePools`, `@EnableCacheServers` is a composite annotation for aggregating multiple `@EnableCacheServer` annotations on a single class. Again, Java 8 and earlier does not allow more than one annotation of the same type to be declared on a single class.

One thing an observant reader may have noticed is that, in all cases, you have specified hard-coded values for all hostnames, ports, and configuration-oriented annotation attributes. This is not ideal when the application gets promoted and deployed to different environments, such as from DEV to QA to STAGING to PROD.

The next section covers how to handle dynamic configuration determined at runtime.

7.4. 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 the configuration attribute could be expressed as an `int` (such as 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 and SpEL expressions in properties or attributes of the configuration metadata when configuring beans in the Spring container. However, this would require all annotation attributes to be of type `String`, thereby giving up type safety, which is not desirable.

So, Spring Data for Pivotal GemFire borrows from another commonly used pattern in Spring, `Configurers`. Many different `Configurer` interfaces are provided in Spring Web MVC, including the `org.springframework.web.servlet.config.annotation.ContentNegotiationConfigurer`.

The `Configurers` design pattern enables application developers to receive a callback to customize the configuration of a component or bean 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 for Pivotal GemFire provides several `Configurer` callback interfaces to customize different aspects of the annotation-based configuration metadata at runtime, before the Spring managed beans that the annotations create are initialized:

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

For example, you can use the `CacheServerConfigurer` and `ClientCacheConfigurer` to customize the port numbers used by your Spring Boot `CacheServer` and `ClientCache` applications, respectively.

Consider the following example from a server application:

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

```
@SpringBootApplication
@CacheServerApplication(name = "SpringServerApplication")
class ServerApplication {

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

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

Next, consider the following example from a client application:

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

```
@SpringBootApplication
@ClientCacheApplication
class ClientApplication {

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

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

By using the provided `Configurers`, you can receive a callback to further customize the configuration that is enabled by the associated annotation at runtime, during startup.

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

All `Configurers` provided by Spring Data for Pivotal GemFire take two bits of information in the callback: the name of the bean created in the Spring container by the annotation and a reference to the `FactoryBean` used by the annotation to create and configure the Pivotal GemFire component (for example, a `ClientCache` instance is created and configured with `ClientCacheFactoryBean`).

SDG `FactoryBeans` are part of the SDG public API and are what you 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 same `FactoryBeans` for their configuration. So, in essence, the annotations are a facade providing an extra layer of abstraction for convenience.

Given that a `Configurer` can be declared as a regular bean definition like any other POJO, you can combine different Spring configuration options, such as the use of Spring Profiles with `Conditions` that use both property placeholders and SpEL expressions. These and other nifty features let you create even more sophisticated and flexible configurations.

However, `Configurers` are not the only option.

7.5. 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.`), which can be declared in a Spring Boot `application.properties` file.

Building on the earlier examples, the client's `application.properties` file would define the following set of properties:

Client `application.properties`

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

The corresponding server's `application.properties` file would define the following properties:

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 you can simplify the `@ClientCacheApplication` class to the following:

Spring `@ClientCacheApplication` class

```
@SpringBootApplication
@ClientCacheApplication
@EnablePools(pools = {
    @EnablePool(name = "Venus"),
    @EnablePool(name = "Saturn"),
    @EnablePool(name = "Neptune")
})
class ClientApplication { .. }
```

Also, the `@CacheServerApplication` class becomes the following:

Spring `@CacheServerApplication` class

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

The preceding example shows why it is important to "name" your annotation-based beans (other than because it is required in certain cases). Doing so makes it possible to reference the bean in the Spring container from XML, properties, and Java. It is even possible to inject annotation-defined beans into an application class, for whatever purpose, as the following example demonstrates:

```
@Component
class MyApplicationComponent {

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

  ...
}
```

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

Oftentimes, an associated annotation attribute property takes two forms: a "named" property along with an "unnamed" property.

The following example shows such an arrangement:

```
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 three named `CacheServers` above, there is also one unnamed `CacheServer` property providing the default value for any unspecified value of 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.

See an annotation's Javadoc for which annotation attributes support property-based configuration and whether they support "named" properties over default, "unnamed" properties.

7.5.1. `Properties` of `Properties`

In the usual Spring fashion, you can even express `Properties` in terms of other `Properties`, whether that is by The following example shows a nested property being set in an `application.properties` file:

Properties of Properties

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

The following example shows a nested property being set in Java:

Property placehodler nesting

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

Property placeholder nesting can be arbitrarily deep.

7.6. Configuring Embedded Services

Pivotal GemFire provides the ability to start many different embedded services that are required by an application, depending on the use case.

7.6.1. Configuring an Embedded Locator

As mentioned previously, Pivotal GemFire Locators are used by clients to connect to and find servers in a cluster. In addition, new members joining an existing cluster use Locators to find their peers.

It is often convenient for application developers as they are developing their Spring Boot and Spring Data for Pivotal GemFire applications to startup up a small cluster of two or three Pivotal GemFire servers. Rather than starting a separate Locator process, you can annotate your Spring Boot `@CacheServerApplication` class with `@EnableLocator`, as follows:

Spring, Pivotal GemFire `CacheServer` application running an embedded Locator

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

The `@EnableLocator` annotation starts an embedded Locator in the Spring Pivotal GemFire `CacheServer` application running on `localhost`, listening on the default Locator port, `10334`. You can customize the `host` (bind address) and `port`

that the embedded Locator binds to by using the corresponding annotation attributes.

Alternatively, you can set the `@EnableLocator` attributes by setting the corresponding `spring.data.gemfire.locator.host` and `spring.data.gemfire.locator.port` properties in `application.properties`.

Then you can start other Spring Boot `@CacheServerApplication`-enabled applications by connecting to this Locator with the following:

Spring, Pivotal GemFire `CacheServer` application connecting to a Locator

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

You can even combine both application classes shown earlier into a single class and use your IDE to create different run profile configurations to launch different instances of the same class with slightly modified configuration by using Java system properties, as follows:

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, you can set and change the following system properties:

IDE run profile configuration

```
spring.data.gemfire.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 set the `-Dspring.profiles.active=embedded-locator` Java system property. Then you can change the `..name` and `..cache.server.port` for each of the other run profiles and have a small cluster (distributed system) of Pivotal GemFire servers running on your local system.

The `@EnableLocator` annotation was meant to be a development-time annotation only and not something an application developer would use in production. We strongly recommend running Locators as standalone, independent processes in the cluster.

More details on how Pivotal GemFire Locators work can be found here.

7.6.2. Configuring an Embedded Manager

A Pivotal GemFire Manager is another peer member or node in the cluster that is responsible for cluster "management". Management involves creating `Regions`, `Indexes`, `DiskStores`, among other things, along with monitoring the runtime operations and behavior of the cluster components.

The Manager lets a JMX-enabled client (such as the *Gfsh* shell tool) connect to the Manager to manage the cluster. It is also possible to connect to a Manager with JDK-provided tools such as JConsole or JVisualVM, given that these are both JMX-enabled clients as well.

Perhaps you would also like to enable the Spring `@CacheServerApplication` shown earlier as a Manager as well. To do so, annotate your Spring `@Configuration` or `@SpringBootApplication` class with `@EnableManager`.

By default, the Manager binds to `localhost`, listening on the default Manager port of `1099`. Several aspects of the Manager can be configured with annotation attributes or the corresponding properties.

The following example shows how to create an embedded Manager in Java:

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 preceding class, you can even use *Gfsh* to connect to the small cluster and manage it, as follows:

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

Monitor and Manage {data-store-name}

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 can connect indirectly to the Manager through the Locator. A Locator lets JMX clients connect and find a Manager in the cluster. If none exists, the Locator assumes the role of a Manager. However, if no Locator exists, we would need to connect directly to the Manager by using the following:

Gfsh `connect` command connecting directly to the Manager

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

Like the `@EnableLocator` annotation, the `@EnableManager` annotation is also meant to be a development-time only annotation and not something an application developer would use in production. We strongly recommend that Managers, like Locators, be standalone, independent and dedicated processes in the cluster.

More details on Pivotal GemFire management and monitoring can be found here.

7.6.3. Configuring the Embedded HTTP Server

Pivotal GemFire 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 Pivotal GemFire'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 Pivotal GemFire-provided web applications, you must have a full installation of Pivotal GemFire installed on your system, and you must set the `GEODE_HOME` environment variable to your installation directory.

To enable the embedded HTTP server, add the `@EnableHttpService` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class, as follows:

Spring `CacheServer` application running the 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 port as needed.

Follow the earlier links for more details on HTTP support and the services provided.

7.6.4. Configuring the Embedded Memcached Server (Gemcached)

Pivotal GemFire also implements the Memcached protocol with the ability to service Memcached clients. That is, Memcached clients can connect to a Pivotal GemFire cluster and perform Memcached operations as if the Pivotal GemFire servers in the cluster were actual Memcached servers.

To enable the embedded Memcached service, add the `@EnableMemcachedServer` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class, as follows:

Spring `CacheServer` application running an embedded Memcached server

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

More details on Pivotal GemFire's Memcached service (called "Gemcached") can be found here.

7.6.5. Configuring the Embedded Redis Server

Pivotal GemFire also implements the Redis server protocol, which enables Redis clients to connect to and communicate with a cluster of Pivotal GemFire servers to issue Redis commands. As of this writing, the Redis server protocol support in Pivotal GemFire is still experimental.

To enable the embedded Redis service, add the `@EnableRedisServer` annotation to any `@PeerCacheApplication` or `@CacheServerApplication` annotated class, as follows:

Spring `CacheServer` *application running an embedded Redis server*

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

More details on Pivotal GemFire's Redis adapter can be found here.

7.7. Configuring Logging

Oftentimes, it is necessary to turn up logging in order to understand exactly what Pivotal GemFire is doing and when.

To enable Logging, annotate your application class with `@EnableLogging` and set the appropriate attributes or associated properties, as follows:

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 (for example, `@ClientCacheApplication(logLevel="info")`), it is easier to customize logging behavior with the `@EnableLogging` annotation.

Additionally, you can configure the `log-level` by setting the `spring.data.gemfire.logging.level` property in `application.properties` .

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

7.8. Configuring Statistics

To gain even deeper insight into Pivotal GemFire at runtime, you can enable statistics. Gathering statistical data facilitates system analysis and troubleshooting when complex problems, which are often distributed in nature and where timing is a crucial factor, occur.

When statistics are enabled, you can use Pivotal GemFire's VSD (Visual Statistics Display) tool to analyze the statistical data that is collected.

To enable statistics, annotate your application class with `@EnableStatistics` , as follows:

Spring `ClientCache` *application with Statistics enabled*

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

Enabling statistics on a server is particularly valuable when evaluating performance. To do so, annotate your `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableStatistics`.

You can 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 Pivotal GemFire's statistics can be found here.

7.9. Configuring PDX

One of the more powerful features of Pivotal GemFire is PDX serialization. While a complete discussion of PDX is beyond the scope of this document, serialization using PDX is a much better alternative to Java serialization, with the following benefits:

- PDX uses a centralized type registry to keep the serialized bytes of an object more compact.
- PDX is a neutral serialization format, allowing both Java and Native clients to operate on the same data set.
- PDX supports versioning and lets object fields be added or removed without affecting existing applications using either older or newer versions of the PDX serialized objects that have changed, without data loss.
- PDX lets object fields be accessed individually in OQL query projections and predicates without the object needing to be de-serialized first.

In general, serialization in Pivotal GemFire is required any time data is transferred to or from clients and servers or between peers in a cluster during normal distribution and replication processes as well as when data is overflowed or persisted to disk.

Enabling PDX serialization is much simpler than modifying all of your application domain object types to implement `java.io.Serializable`, especially when it may be undesirable to impose such restrictions on your application domain model or you do not have any control over the objects your are serializing, which is especially true when using a 3rd party library (e.g. think of a geo-spatial API with `Coordinate` types).

To enable PDX, annotate your application class with `@EnablePdx`, as follows:

Spring `ClientCache` application with PDX enabled

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

Typically, an application's domain object types either implements the `org.apache.geode.pdx.PdxSerializable` interface or you can implement and register a non-invasive implementation of the `org.apache.geode.pdx.PdxSerializer` interface to handle all the application domain object types that need to be serialized.

Unfortunately, Pivotal GemFire only lets one `PdxSerializer` be registered, which suggests that all application domain object types need to be handled by a single `PdxSerializer` instance. However, that is a serious anti-pattern and an unmaintainable practice.

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

By using the Composite Software Design Pattern, you can provide an implementation of the `PdxSerializer` interface that aggregates all of the 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 container and refer to this composite `PdxSerializer` by its bean name in the `@EnablePdx` annotation using the `serializerBeanName` attribute. Spring Data for Pivotal GemFire takes care of registering it with Pivotal GemFire on your behalf.

The following example shows how to create a custom composite `PdxSerializer`:

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 Pivotal GemFire's `org.apache.geode.pdx.ReflectionBasedAutoSerializer` as a bean definition in a Spring context.

Alternatively, you should use Spring Data for Pivotal GemFire's more robust `org.springframework.data.gemfire.mapping.MappingPdxSerializer`, which uses Spring Data mapping metadata 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.

7.10. Configuring Pivotal 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 Pivotal GemFire properties are still accessible from the `@EnableGemFireProperties` annotation.

Annotating your application class with `@EnableGemFireProperties` is convenient and a nice alternative to creating a `gemfire.properties` file or setting Pivotal GemFire properties as Java system properties on the command line when launching your application.

We recommend that these Pivotal GemFire properties be set in a `gemfire.properties` file when deploying your application to production. However, at development time, it can be convenient to set these properties individually, as needed, for prototyping, debugging and testing purposes.

A few examples of some of the less common Pivotal GemFire properties that you usually need not worry about include, but are not limited to: `ack-wait-threshold`, `disable-tcp`, `socket-buffer-size`, and others.

To individually set any Pivotal GemFire property, annotate your application class with `@EnableGemFireProperties` and set the Pivotal GemFire properties you want to change from the default value set by Pivotal GemFire with the corresponding attribute, as follows:

Spring `ClientCache` application with specific Pivotal GemFire properties set

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

Keep in mind that some of the Pivotal GemFire properties are client-specific (for example, `conflateEvents`), while others are server-specific (for example `distributedSystemId`, `enableNetworkPartitionDetection`, `enforceUniqueHost`, `memberTimeout`, `redundancyZone`, and others).

More details on Pivotal GemFire properties can be found here.

7.11. Configuring Regions

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

Pivotal GemFire 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, which makes it more conducive for building effective indexes and writing queries. We cover indexing later.

Previously, Spring Data for Pivotal GemFire users needed to explicitly define and declare the Regions used by their applications to store data by writing very verbose Spring configuration metadata, whether using SDG's `FactoryBeans` from the API with Spring's Java-based container configuration or using XML.

The following example demonstrates how to configure a Region bean in Java:

Example Region bean definition using Spring's Java-based container configuration

```
@Configuration
class GemFireConfiguration {

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

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

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

      return exampleRegion;
  }

  ...
}
```

The following example demonstrates how to configure the same Region bean in XML:

Example Region bean definition using SDG's XML Namespace

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

While neither Java nor XML configuration is all that difficult to specify, either one can be cumbersome, especially if an application requires a large number of Regions. Many relational database-based applications can have hundreds or even thousands of tables.

Defining and declaring all these Regions by hand would be cumbersome and error prone. Well, now there is a better way.

Now you can define and configure Regions based on their application domain objects (entities) themselves. No longer do you need to explicitly define `Region` bean definitions in Spring configuration metadata, unless you require finer-grained control.

To simplify Region creation, Spring Data for Pivotal GemFire combines the use of Spring Data Repositories with the expressive power of annotation-based configuration using the new `@EnableEntityDefinedRegions` annotation.

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

First, an application developer starts by defining the application's domain objects (entities), as follows:

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, you define a basic repository for `Books` by extending Spring Data Commons `org.springframework.data.repository.CrudRepository` interface, as follows:

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 (such as `findById(..)`). You can define additional, more sophisticated queries by declaring query methods on the repository interface (for example, `List<BooK> findByAuthor(Author author);`).

Under the hood, Spring Data for Pivotal GemFire provides an implementation of your application's repository interfaces when the Spring container is bootstrapped. SDG even implements the query methods you define so long as you follow the conventions.

Now, when you defined the `Book` class, you also specified the Region in which instances of `Book` are mapped (stored) by declaring the Spring Data for Pivotal GemFire mapping annotation, `@Region` on the entity's type. Of course, if the entity type (`Book`, in this case) referenced in the type parameter of the repository interface (`BookRepository`, in this case) is not annotated with `@Region`, the name is derived from the simple class name of the entity type (also `Book`, in this case).

Spring Data for Pivotal GemFire uses the mapping context, which contains mapping metadata for all the entities defined in your application, to determine all the Regions that are needed at runtime.

To enable and use this feature, annotate the application class with `@EnableEntityDefinedRegions`, as follows:

Entity-defined Region Configuration

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

Creating Regions from entity classes is most useful when using Spring Data Repositories in your application. Spring Data for Pivotal GemFire's Repository support is enabled with the `@EnableGemfireRepositories` annotation, as shown in the preceding example.

Currently, only entity classes explicitly annotated with `@Region` are picked up by the scan and will have Regions created. If an entity class is not explicitly mapped with `@Region` no Region will be created.

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

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, you 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 that contains the entity's class, or by using a non-entity placeholder class specifically created for identifying the package to scan.

The following example shows how to specify the entity types to scan:

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

See the `@EnableEntityDefinedRegions` annotation Javadoc for more details.

7.11.1. Configuring Type-specific Regions

Pivotal GemFire 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 (i.e. distributed, replicated, and so on).

Other configuration settings (such as the Region's `scope`) can also affect how data is managed. See "Storage and Distribution Options" in the Pivotal GemFire User Guide for more details.

When you annotate your application domain object types with the generic `@Region` mapping annotation, Spring Data for Pivotal GemFire decides 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 you declare the application as a `ClientCache` by using the `@ClientCacheApplication` annotation, SDG creates a client `PROXY Region` by default. Alternatively, if you declare the application as a peer `Cache` by using either the `@PeerCacheApplication` or `@CacheServerApplication` annotations, SDG creates a server `PARTITION Region` by default.

Of course, you can always override the default when necessary. To override the default applied by Spring Data for Pivotal GemFire, four new Region mapping annotations have been introduced:

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

The `@ClientRegion` mapping annotation is specific to client applications. All of the other Region mapping annotations listed above can only be used in server applications that have an embedded peer `Cache`.

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 on the user's behalf. In this case, the data does not need to be distributed back to the server 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. (See "Configuring Expiration" for more on expiration policies.)

Region-level Idle-Timeout (TTI) and Time-To-Live (TTL) expiration policies are independent of and different from entry-level TTI and TTL expiration policies.

In any case, if you want to create a local-only client Region where the data is not going to be distributed back to a corresponding Region on the server with the same name, you can declare the `@ClientRegion` mapping annotation and set the `shortcut` attribute to `ClientRegionShortcut.LOCAL`, as follows:

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. For example, the `collocatedWith` and `redundantCopies` attributes in the `PartitionRegion` annotation apply to server-side, `PARTITION` Regions only.

More details on Pivotal GemFire Region types can be found here.

7.11.2. Configured Cluster-defined Regions

In addition to the `@EnableEntityDefinedRegions` annotation, Spring Data for Pivotal GemFire also provides the inverse annotation, `@EnableClusterDefinedRegions`. Rather than basing your Regions on the entity classes defined and driven from your application use cases (UC) and requirements (the most common and logical approach), alternatively, you can declare your Regions from the Regions already defined in the cluster to which your `ClientCache` application will connect.

This allows you to centralize your configuration using the cluster of servers as the primary source of data definitions and ensure that all client applications of the cluster have a consistent configuration. This is particularly useful when quickly scaling up a large number instances of the same client application to handle the increased load in a cloud-managed environment.

The idea is, rather than the client application(s) driving the data dictionary, the user defines Regions using Pivotal GemFire's *Gfsh* CLI shell tool. This has the added advantage that when additional peers are added to the cluster, they too will also have and share the same configuration since it is remembered by Pivotal GemFire's *Cluster Configuration Service*.

By way of example, a user might defined a Region in *Gfsh*, as follows:

Defining a Region with Gfsh

```
gfsh>create region --name=Books --type=PARTITION
 Member   | Status
--------- | ---------------------------------------
ServerOne | Region "/Books" created on "ServerOne"
ServerTwo | Region "/Books" created on "ServerTwo"

gfsh>list regions
List of regions
---------------
Books

gfsh>describe region --name=/Books
..........................................................
Name            : Books
Data Policy     : partition
Hosting Members : ServerTwo
                  ServerOne

Non-Default Attributes Shared By Hosting Members
```

```
 Type   |    Name      | Value
------ | ------------ | ---------
Region | size         | 0
       | data-policy  | PARTITION
```

With Pivotal GemFire's *Cluster Configuration Service*, any additional peer members added to the cluster of servers to handle the increased load (on the backend) will also have the same configuration, for example:

Adding an additional peer member to the cluster

```
gfsh>list members
  Name     | Id
---------- | ---------------------------------------------
Locator    | 10.0.0.121(Locator:68173:locator)<ec><v0>:1024
ServerOne  | 10.0.0.121(ServerOne:68242)<v3>:1025
ServerTwo  | 10.0.0.121(ServerTwo:68372)<v4>:1026

gfsh>start server --name=ServerThree --log-level=config --server-port=41414
Starting a Geode Server in /Users/you/geode/cluster/ServerThree...
...
Server in /Users/you/geode/cluster/ServerThree... on 10.0.0.121[41414] as ServerThree is currently online.
Process ID: 68467
Uptime: 3 seconds
Geode Version: 1.2.1
Java Version: 1.8.0_152
Log File: /Users/you/geode/cluster/ServerThree/ServerThree.log
JVM Arguments: -Dgemfire.default.locators=10.0.0.121[10334]
  -Dgemfire.use-cluster-configuration=true
  -Dgemfire.start-dev-rest-api=false
  -Dgemfire.log-level=config
  -XX:OnOutOfMemoryError=kill -KILL %p
  -Dgemfire.launcher.registerSignalHandlers=true
  -Djava.awt.headless=true
  -Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /Users/you/geode/cluster/apache-geode-1.2.1/lib/geode-core-1.2.1.jar
  :/Users/you/geode/cluster/apache-geode-1.2.1/lib/geode-dependencies.jar

gfsh>list members
   Name      | Id
-----------  | ---------------------------------------------
Locator      | 10.0.0.121(Locator:68173:locator)<ec><v0>:1024
ServerOne    | 10.0.0.121(ServerOne:68242)<v3>:1025
ServerTwo    | 10.0.0.121(ServerTwo:68372)<v4>:1026
ServerThree  | 10.0.0.121(ServerThree:68467)<v5>:1027

gfsh>describe member --name=ServerThree
Name         : ServerThree
Id           : 10.0.0.121(ServerThree:68467)<v5>:1027
Host         : 10.0.0.121
Regions      : Books
PID          : 68467
Groups       :
Used Heap    : 37M
Max Heap     : 3641M
Working Dir  : /Users/you/geode/cluster/ServerThree
Log file     : /Users/you/geode/cluster/ServerThree/ServerThree.log
Locators     : 10.0.0.121[10334]

Cache Server Information
Server Bind              :
Server Port              : 41414
Running                  : true
Client Connections       : 0
```

As you can see, "ServerThree" now has the "Books" Region. If the any or all of the server go down, they will have the same configuration along with the "Books" Region when they come back up.

On the client-side, many Book Store client application instances might be started to process books against the Book Store online service. The "Books" Region might be 1 of many different Regions needed to implement the Book Store application service. Rather than have to create and configure each Region individually, SDG conveniently allows the client application Regions to be defined from the cluster, as follows:

Defining Client Regions from the Cluster with `@EnableClusterDefinedRegions`

```
@ClientCacheApplication
@EnableClusterDefinedRegions
class BookStoreClientApplication {

    public static void main(String[] args) {
        ....
    }

    ...
}
```

`@EnableClusterDefinedRegions` can only used on the client.

You can use the `clientRegionShortcut` annotation attribute to control the type of Region created on the client. By default, a client `PROXY` Region is created. Set `clientRegionShortcut` to `ClientRegionShortcut.CACHING_PROXY` to implement "*near caching*". This setting applies to all client Regions created from Cluster-defined Regions. If you want to control individual settings (like data policy) of the client Regions created from Regions defined on the Cluster, then you can implement a `RegionConfigurer` with custom logic based on the Region name.

Then, it becomes a simple matter to use the "Books" Region in your application. You can inject the "Books" Region directly, as follows:

Using the "Books" Region

```
@org.springframework.stereotype.Repository
class BooksDataAccessObject {

    @Resource(name = "Books")
    private Region<ISBN, Book> books;

    // implement CRUD and queries with the "Books" Region
}
```

Or, even define a Spring Data Repository definition based on the application domain type (entity), `Book`, mapped to the "Books" Region, as follows:

Using the "Books" Region with a SD Repository

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

```
}
```

You can then either inject your custom `BooksDataAccessObject` or the `BookRepository` into your application service components to carry out whatever business function required.

7.11.3. Configuring Eviction

Managing data with Pivotal GemFire is an active task. Tuning is generally required, and you must employ a combination of features (for example, both eviction and expiration) to effectively manage your data in memory with Pivotal GemFire.

Given that Pivotal GemFire 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), you can increase capacity by adding new nodes to the cluster. This is commonly referred to as linear scale-out (rather than scaling up, which means adding more memory, more CPU, more disk, or 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 venturing near 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, Pivotal GemFire supports Least Recently Used (LRU) eviction. That is, Pivotal GemFire evicts Region entries based on when those entries were last accessed by using the Least Recently Used algorithm.

To enable eviction, annotate the application class with `@EnableEviction`, as follows:

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 servers.

As shown earlier, the `policies` attribute can specify one or more nested `@EvictionPolicy` annotations, with each one being individually catered to one or more Regions where the eviction policy needs to be applied.

Additionally, you can reference a custom implementation of Pivotal GemFire's `org.apache.geode.cache.util.ObjectSizer` interface, which can be defined as a bean in the Spring container and referenced by name by using the `objectSizerName` attribute.

An `ObjectSizer` lets you define the criteria used to evaluate and determine the the size of objects stored in a Region.

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

More details on Pivotal GemFire eviction can be found here.

7.11.4. Configuring Expiration

Along with eviction, expiration can also be used to manage memory by allowing entries stored in a Region to expire. Pivotal GemFire supports both Time-to-Live (TTL) and Idle-Timeout (TTI) entry expiration policies.

Spring Data for Pivotal GemFire's annotation-based expiration configuration is based on the earlier and existing entry expiration annotation support added in Spring Data for Pivotal GemFire version 1.5.

Essentially, Spring Data for Pivotal GemFire's expiration annotation support is based on a custom implementation of Pivotal GemFire's `org.apache.geode.cache.CustomExpiry` interface. This `o.a.g.cache.CustomExpiry` implementation inspects the user's application domain objects stored in a Region for the presence of type-level expiration annotations.

Spring Data for Pivotal GemFire provides the following expiration annotations:

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

An application domain object type can be annotated with one or more of the expiration annotations, as follows:

Applicaton domain object specific expiration policy

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

To enable expiration, annotate the application class with `@EnableExpiration`, as follows:

Spring application with expiration enabled

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

In addition to application domain object type-level expiration policies, you can directly and individually configure expiration policies on a Region by Region basis using the `@EnableExpiration` annotation, as follows:

Spring application with region-specific 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 { .. }
```

The preceding example sets expiration policies for the `Books`, `Customers`, and `Orders` Regions.

Expiration policies are usually set on the Regions in the servers.

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

More details on Pivotal GemFire expiration can be found here.

7.11.5. Configuring Compression

In addition to eviction and expiration, you can also configure your data Regions with compression to reduce memory consumption.

Pivotal GemFire lets you compress in memory Region values by using pluggable `Compressors`, or different compression codecs. Pivotal GemFire uses Google's Snappy compression library by default.

To enable compression, annotate the application class with `@EnableCompression`, as follows:

Spring application with Region compression enabled

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

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

The `compressorBeanName` defaults to `SnappyCompressor`, enabling Pivotal GemFire's `SnappyCompressor`.

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

Alternatively, you can 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.

To use Pivotal GemFire's Region compression feature, you must include the `org.iq80.snappy:snappy` dependency in your application's `pom.xml` file (for Maven) or `build.gradle` file (for Gradle). This is necessary only if you use Pivotal GemFire's default support for Region compression, which uses the `SnappyCompressor` by default. Of course, if you use another compression library, you need to include dependencies for that compression library on your application's classpath. Additionally, you need to implement Pivotal GemFire's `Compressor` interface to adapt your compression library of choice, define it as a bean in the Spring compressor, and set the `compressorBeanName` to this custom bean definition.

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

More details on Pivotal GemFire compression can be found here.

7.11.6. Configuring Off-Heap Memory

Another effective means for reducing pressure on the JVM's Heap memory and minimizing GC activity is to use Pivotal GemFire's off-heap memory support.

Rather than storing Region entries on the JVM Heap, entries are stored in the system's main memory. Off-heap memory generally works best when the objects being stored are uniform in size, are mostly less than 128K, and do not need to be

deserialized frequently, as explained in the Pivotal GemFire User Guide.

To enable off-heap, annotate the application class with `@EnableOffHeap`, as follows:

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 can use in either megabytes (`m`) or gigabytes (`g`).

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

Alternatively, you can 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.

7.11.7. Configuring Disk Stores

Alternatively, you can configure Regions to persist data to disk. You can also configure Regions to overflow data to disk when Region entries are evicted. In both cases, a `DiskStore` is required to persist and/or overflow the data. When an explicit `DiskStore` has not been configured for a Region with persistence or overflow, Pivotal GemFire uses the `DEFAULT DiskStore`.

We recommend defining Region-specific `DiskStores` when persisting and/or overflowing data to disk.

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

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

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

Using the `@EnableDiskStore` annotation, you 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 one `DiskStore` can be defined by using the composite, `@EnableDiskStores` annotation.

As with other annotations in Spring Data for Pivotal GemFire's annotation-based configuration model, both `@EnableDiskStore` and `@EnableDiskStores` have many attributes along with associated configuration properties to customize the `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 override a particular global setting, but the `@EnableDiskStores` annotation conveniently defines common configuration attributes that apply across all `DiskStores` aggregated by the annotation.

Spring Data for Pivotal GemFire also provides the `DiskStoreConfigurer` callback interface, which can be declared in Java configuration and used instead of configuration properties to customize a `DiskStore` at runtime, as the following example shows:

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 Pivotal GemFire Region persistence and overflow (using DiskStores) can be found here.

7.11.8. Configuring Indexes

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

In addition to `Region.get(key)` operations, particularly when the key is known in advance, data is commonly retrieved by executing queries on the Regions that contain the data. With Pivotal GemFire, queries are written by using the Object Query Language (OQL), and the specific data set that a client wishes to access is expressed in the query's predicate (for example, `SELECT * FROM /Books b WHERE b.author.name = 'Jon Doe'`).

Generally, querying without indexes is inefficient. When executing queries without an index, Pivotal GemFire 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 interest, as expressed by the query's projection. Different types of indexes, such as key and hash indexes, can be created.

Spring Data for Pivotal GemFire makes it easy to create indexes on Regions where the data is stored and accessed. Rather than explicitly declaring `Index` bean definitions by using Spring config as before, we can create an `Index` bean definition in Java, as follows:

Index bean definition using Java config

```
@Bean("BooksIsbnIndex")
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;
}
```

Alternatively, we can use XML to create an `Index` bean definition, as follows:

Index bean definition using XML

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

However, now you can directly define indexes on the fields of your application domain object types for which you know will be used in query predicates to speed up those queries. You can even apply indexes for OQL queries generated from user-defined query methods on an application's repository interfaces.

Re-using the example `Book` entity class from earlier, we can annotate the fields on `Book` that we know are used in queries that we define with query methods in the `BookRepository` interface, as follows:

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 for Pivotal GemFire, the `@Id` annotated field or property is used as the key in the Region when storing the entry.

- `@Id` annotated fields or properties result in the creation of an Pivotal GemFire `KEY` Index.
- `@Indexed` annotated fields or properties result in the creation of an Pivotal GemFire `HASH` Index (the default).
- `@LuceneIndexed` annotated fields or properties result in the creation of an Pivotal GemFire Lucene Index, used in text-based searches with Pivotal GemFire'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 or property of the class 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 domain object's class, or the simple name of the domain object class if the `@Region` annotation was not specified.

Of course, you can explicitly set any of the `@Indexed` annotation attributes to override the default values provided by Spring Data for Pivotal GemFire.

Application domain object type modeling a Book with customized 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 container for the index. If necessary, this index bean can even be injected by name into another application component.

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

To enable indexing, annotate the application class with `@EnableIndexing`, as follows:

Spring application with Indexing enabled

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

> The `@EnablingIndexing` annotation has no effect unless the `@EnableEntityDefinedRegions` is also

declared. Essentially, indexes are defined from fields or properties on the entity class types, and entity classes must be scanned to inspect the entity's fields and properties for the presence of index annotations. Without this scan, index annotations cannot be found. We also strongly recommend that you limit the scope of the scan.

While Lucene queries are not (yet) supported on Spring Data for Pivotal GemFire repositories, SDG does provide comprehensive support for Pivotal GemFire Lucene queries by using the familiar Spring template design pattern.

Finally, we close this section with a few extra tips to keep in mind when using indexes:

- While OQL indexes are not required to execute OQL Queries, Lucene Indexes are required to execute Lucene text-based searches.
- OQL indexes are not persisted to disk. They are only maintained in memory. So, when an Pivotal GemFire node is restarted, the index must be rebuilt.
- 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 you can use when restarting your Spring application where indexes have to be rebuilt is to first define all the indexes up front and then create them all at once, which, in Spring Data for Pivotal GemFire, happens when the Spring container is refreshed.

You can define indexes up front and then create them all at once by setting the `define` attribute on the `@EnableIndexing` annotation to `true`.

See "Creating Multiple Indexes at Once" in Pivotal GemFire's User Guide for more details.

Creating sensible indexes is an important task, since it is possible for a poorly designed index to do more harm than good.

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

More details on Pivotal GemFire OQL queries can be found here.

More details on Pivotal GemFire indexes can be found here.

More details on Pivotal GemFire Lucene queries can be found here.

7.12. Configuring Continuous Queries

Another very important and useful feature of Pivotal GemFire is Continuous Queries.

In a world of Internet-enabled things, events and streams of data come from everywhere. Being able to handle and process a large stream of data and react to events in real time is 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 applications.

Fortunately, Pivotal GemFire was ahead of its time in this regard. By 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 occur. The data that a client application may be interested in is expressed as an OQL query, where the query predicate is used to filter or identify the data of interest. 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 for Pivotal GemFire makes it easy to define and register CQs, along with an associated listener to handle and process CQ events without all the cruft of Pivotal GemFire's plumbing. SDG's new annotation-based configuration for CQs builds on the existing Continuous Query support in the continuous query listener container.

For instance, say a book publisher wants to register interest in and receive notification any time 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, annotate your application class with `@EnableContinuousQueries`.

Defining Continuous Queries consists of annotating any Spring `@Component`-annotated POJO class methods with the `@ContinuousQuery` annotation (in similar fashion to SDG's Function-annotated POJO methods). A POJO method defined with a CQ by using the `@ContinuousQuery` annotation is called any time data matching the query predicate is added or changed.

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

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

More details on Spring Data for Pivotal GemFire's continuous query support can be found here.

More details on Pivotal GemFire's Continuous Queries can be found here.

7.13. Configuring Spring's Cache Abstraction

With Spring Data for Pivotal GemFire, Pivotal GemFire can be used as a caching provider in Spring's cache abstraction.

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

In Spring Data for Pivotal GemFire, a Spring `Cache` corresponds directly to a Pivotal GemFire Region. The Region must exist before any caching annotated application service methods are called. This is true for any of Spring's caching annotations (that is, `@Cacheable`, `@CachePut` and `@CacheEvict`) that identify the cache to use in the service 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, as the following example shows:

```
@Service
class PointOfSaleService

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

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

  ...
}
```

To make your work easier when you use Spring Data for Pivotal GemFire with Spring's Cache Abstraction, two new features have been added to the annotation-based configuration model.

Consider the following Spring caching configuration:

Enabling Caching using Pivotal GemFire as the caching provider

```
@EnableCaching
class CachingConfiguration {

  @Bean
  GemfireCacheManager cacheManager(GemFireCache gemfireCache) {

      GemfireCacheManager cacheManager = new GemfireCacheManager();

      cacheManager.setCache(gemfireCache);

      return cacheManager;
  }

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

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

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

    return bookPricesRegion;
  }

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

Using Spring Data for Pivotal GemFire's new features, you can simplify the same caching configuration to the following:

Enabling Pivotal GemFire Caching

```
@EnableGemfireCaching
@EnableCachingDefinedRegions
```

```
class CachingConfiguration {

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

First, the `@EnableGemfireCaching` annotation replaces both the Spring `@EnableCaching` annotation and the need to declare an explicit `CacheManager` bean definition (named "cacheManager") in the Spring config.

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

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

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

SDG also supports JCache (JSR-107) cache annotations on application service components. See the core *Spring Framework Reference Guide* for the equivalent Spring caching annotation to use in place of JCache caching annotations.

Refer to the "Support for the Spring Cache Abstraction" section for more details on using Pivotal GemFire as a caching provider in Spring's Cache Abstraction.

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

7.14. Configuring Cluster Configuration Push

This may be the most exciting new feature in Spring Data for Pivotal GemFire.

When a client application class is annotated with `@EnableClusterConfiguration`, any Regions or indexes defined and declared as beans in the Spring container 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 Pivotal GemFire remembers the configuration pushed by the client when using HTTP. If all the nodes in the cluster go down, they come back up with the same configuration as before. If a new server is added to the cluster, it will acquire identical configuration.

In a sense, this feature is not much different than if you were to use *Gfsh* to manually create the Regions and indexes on all the servers in the cluster. Except that now, with Spring Data for Pivotal GemFire, you no longer need to use *Gfsh* to create Regions and indexes. Your Spring Boot application, enabled with the power of Spring Data for Pivotal GemFire, already contains all the configuration metadata needed to create Regions and indexes for you.

When you use the Spring Data Repository abstraction, we know all the Regions (such as those defined by the `@Region` annotated entity classes) and indexes (such as those defined by the `@Indexed` -annotated entity fields and properties) that your application will need.

When you use Spring's Cache Abstraction, we also know all the Regions for all the caches identified in the caching annotations needed by the application's service components.

Essentially, you are already telling us everything we need to know simply by developing your application with the Spring Framework by using all of its provided services, infrastructure, and other components, whether expressed in annotation metadata, Java, XML or otherwise, and whether for configuration, mapping, or whatever the purpose.

The point is that you can focus on your application's business logic while using the framework's services and supporting infrastructure (such as Spring's Cache Abstraction, Spring Data Repositories, Spring's Transaction Management, and so on) and Spring Data for Pivotal GemFire takes care of all the Pivotal GemFire plumbing required by those framework services on the your 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 Pivotal GemFire's Cluster Configuration service. Pivotal GemFire's Cluster Configuration service is also the same service used by *Gfsh* to record schema-related changes (for example, `gfsh> create region --name=Example --type=PARTITION`) issued by the user to the cluster from the shell.

Of course, since the cluster may "remember" the prior configuration pushed by a client from a previous run, Spring Data for Pivotal GemFire is careful not to stomp on any existing Regions and indexes already defined in the servers. This is especially important, for instance, when Regions already contain data.

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

Unlike *Gfsh*, Spring Data for Pivotal GemFire supports the creation of Regions and indexes only on the servers from a client. For advanced configuration and use cases, you should use *Gfsh* to manage the cluster.

Consider the power expressed in the following configuration:

Spring `ClientCache` *application*

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

You instantly get a Spring Boot application with a Pivotal GemFire `ClientCache` instance, Spring Data Repositories, Spring's Cache Abstraction with Pivotal GemFire as the caching provider (where Regions and indexes are not only created on the client but pushed to the servers in the cluster).

From there, you only need to do the following:

- Define the application's domain model objects annotated with mapping and index annotations.
- Define Repository interfaces to support basic data access operations and simple queries for each of the entity types.
- Define the service components containing the business logic transacting the entities.
- Declare the appropriate annotations on service methods that require caching, transactional behavior, and so on.

Nothing in this case pertains to the infrastructure and plumbing required in the application's back-end services (such as Pivotal GemFire). Database users have similar features. Now Spring and Pivotal GemFire developers can, too.

When combined with the following Spring Data for Pivotal GemFire annotations, this application really starts to take flight, with very little effort:

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

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

7.15. 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 by using the Secure Sockets Extension (SSE) and Transport Layer Security (TLS).

To enable SSL, annotate your application class with `@EnableSsl`, as follows:

Spring `ClientCache` application with SSL enabled

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

Then you need to set the necessary SSL configuration attributes or properties: keystores, usernames/passwords, and so on.

You can individually configure different Pivotal GemFire components (`GATEWAY`, `HTTP`, `JMX`, `LOCATOR`, and `SERVER`) with SSL, or you can collectively configure them to use SSL by using the `CLUSTER` enumerated value.

You can specify which Pivotal GemFire components the SSL configuration settings should applied by using the nested `@EnableSsl` annotation, `components` attribute with enumerated values from the `Component` enum, as follows:

Spring `ClientCache` application with SSL enabled by component

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

In addition, you can also specify component-level SSL configuration (`ciphers` , `protocols` and `keystore` / `truststore` information) by using the corresponding annotation attribute or associated configuration properties.

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

More details on Pivotal GemFire SSL support can be found here.

7.16. Configuring Security

Without a doubt, application security is extremely important, and Spring Data for Pivotal GemFire provides comprehensive support for securing both Pivotal GemFire clients and servers.

Recently, Pivotal GemFire 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 to enforce security.

The remainder of this section demonstrates how Spring Data for Pivotal GemFire can simplify Pivotal GemFire's security story even further.

7.16.1. Configuring Server Security

There are several different ways in which you can configure security for servers in a Pivotal GemFire cluster.

- Implement the Pivotal GemFire `org.apache.geode.security.SecurityManager` interface and set Pivotal GemFire's `security-manager` property to refer to your application `SecurityManager` implementation using the fully qualified class name. Alternatively, users can construct and initialize an instance of their `SecurityManager` implementation and set it with the CacheFactory.setSecurityManager(:SecurityManager) method when creating a Pivotal GemFire peer `Cache` .
- Create an Apache Shiro `shiro.ini` file with the users, roles, and permissions defined for your application and then set the Pivotal GemFire `security-shiro-init` property to refer to this `shiro.ini` file, which must be available in the `CLASSPATH` .
- Using only Apache Shiro, annotate your Spring Boot application class with Spring Data for Pivotal GemFire's new `@EnableSecurity` annotation and define one or more Apache Shiro `Realms` as beans in the Spring container for accessing your application's security metadata (that is, authorized users, roles, and permissions).

The problem with the first approach is that you must implement your own `SecurityManager` , which can be quite tedious and error-prone. Implementing a custom `SecurityManager` offers some flexibility in accessing security metadata from whatever data source stores the metadata, such as LDAP or even a proprietary, internal data source. However, that problem has already been solved by configuring and using Apache Shiro `Realms` , which is more universally known and non-Pivotal GemFire-specific.

See Pivotal GemFire's security examples for Authentication and Authorization as one possible way to implement your own custom, application-specific `SecurityManager` . However, we strongly recommend **against** doing so.

The second approach, using an Apache Shiro INI file, is marginally better, but you still need 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 (that is, Apache Shiro's Security framework) and is easy to setup, as the following example shows:

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 GemFirePermissionResolver());

      return propertiesRealm;
  }
}
```

The configured `Realm` shown in the preceding example could easily have been any of Apache Shiro's supported `Realms`:

- ActiveDirectory
- JDBC
- JNDI
- LDAP
- A `Realm` supporting the INI format.

You could even create a custom implementation of an Apache Shiro `Realm`.

See Apache Shiro's documentation on Realms for more details.

When Apache Shiro is on the `CLASSPATH` of the servers in the cluster and one or more Apache Shiro `Realms` have been defined as beans in the Spring container, Spring Data for Pivotal GemFire detects this configuration and uses Apache Shiro as the security provider to secure your Pivotal GemFire servers when the `@EnableSecurity` annotation is used.

You can find more information about Spring Data for Pivotal GemFire's support for Pivotal GemFire's new integrated security framework using Apache Shiro in this spring.io blog post.

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

More details on Pivotal GemFire security can be found here.

7.16.2. Configuring Client Security

The security story would not be complete without discussing how to secure Spring-based, Pivotal GemFire cache client applications as well.

Pivotal GemFire's process for securing a client application is, honestly, rather involved. In a nutshell, you need to:

1. Provide an implementation of the `org.apache.geode.security.AuthInitialize` interface.
2. Set the Pivotal GemFire `security-client-auth-init` (System) property to refer to the custom, application-provided `AuthInitialize` interface.
3. Specify the user credentials in a proprietary, Pivotal GemFire `gfsecurity.properties` file.

Spring Data for Pivotal GemFire simplifies all of those steps by using the same `@EnableSecurity` annotation that was used in the server applications. In other words, the same `@EnableSecurity` annotation handles security for both client and server applications. This feature 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 you need to do on the client is the following:

Spring client application using `@EnableSecurity`

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

Then you can define the familiar Spring Boot `application.properties` file containing the required username and password, as the following example shows, 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
```

By default, Spring Boot can find your `application.properties` file when it is placed in the root of the application's `CLASSPATH`. Of course, Spring supports many ways to locate resources by using its Resource abstraction.

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

More details on Pivotal GemFire Security can be found here.

7.17. Configuration Tips

The following tips can help you get the most out of using the new annotation-based configuration model:

- Configuration Organization
- Additional Configuration-based Annotations

7.17.1. Configuration Organization

As we saw in the section on "Configuring Cluster Configuration Push", when many Pivotal GemFire or Spring Data for Pivotal GemFire features are enabled by using annotations, we begin 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, consider the following declaration:

Spring `ClientCache` *application with the kitchen sink*

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

We could break this configuration down by concern, as follows:

Spring `ClientCache` *application with the kitcken sink to boot*

```
@SpringBootApplication
@Import({ GemFireConfiguration.class, CachingConfiguration.class,
    FunctionsConfiguration.class, QueriesConfiguration.class,
    RepositoriesConfiguration.class })
class ClientApplication { .. }

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

@EnableGemfireCaching
@EnableCachingDefinedRegions
class CachingConfiguration { .. }

@EnableGemfireFunctionExecutions
class FunctionsConfiguration { .. }

@EnableContinuousQueries
class QueriesConfiguration {

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

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

While it does not matter to the Spring Framework, we generally recommend aiming for readability, for the sake of the next person who has to maintain the code (which might be you at some point in the future).

7.17.2. Additional Configuration-based Annotations

The following SDG Annotations were not discussed in this reference documentation, either because the annotation supports a deprecated feature of Pivotal GemFire or because there are better, alternative ways to accomplishing the function that the annotation provides:

- `@EnableAuth` : Enables Pivotal GemFire's old authentication and authorization security model. (Deprecated. Pivotal GemFire's new integrated security framework can be enabled on both clients and servers by using SDG's `@EnableSecurity` annotation, as described in "Configuring Security".)
- `@EnableAutoRegionLookup` : Not recommended. Essentially, this annotation supports finding Regions defined in external configuration metadata (such as `cache.xml` or Cluster Configuration when applied to a server) and automatically registers those Regions as beans in the Spring container. This annotation corresponds with the `<gfe:auto-region-lookup>` element in SDG's XML namespace. More details can found here. Users should generally prefer Spring configuration when using Spring and Spring Data for Pivotal GemFire. See "Configuring Regions" and "Configuring Cluster Configuration Push" instead.
- `@EnableBeanFactoryLocator` : Enables the SDG `GemfireBeanFactoryLocator` feature, which is only useful when using external configuration metadata (for example, `cache.xml`). For example, if you define a `CacheLoader` on a Region defined in `cache.xml` , you can still autowire this `CacheLoader` with, say, a relational database `DataSource` bean defined in Spring configuration. This annotation takes advantage of this SDG feature and might be useful if you have a large amount of legacy configuration metadata, such as `cache.xml` files.
- `@EnableGemFireAsLastResource` : Discussed in Global - JTA Transaction Management with Pivotal GemFire.
- `@EnableMcast` : Enables Pivotal GemFire's old peer discovery mechanism that uses UDP-based multi-cast networking. (*Deprecated*. Use Pivotal GemFire Locators instead. See "Configuring an Embedded Locator".
- `@EnableRegionDataAccessTracing` : Useful for debugging purposes. This 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 container, intercepting the Region operation and logging the event.

7.18. Conclusion

As we learned in the previous sections, Spring Data for Pivotal GemFire's new annotation-based configuration model provides a tremendous amount of power. Hopefully, it lives up to its goal of making it easier for you to *get started quickly* and *easily* when using Pivotal GemFire with Spring.

Keep in mind that, when you use the new annotations, you can still use Java configuration or XML configuration. You can even combine all three approaches by using Spring's `@Import` and `@ImportResource` annotations on a Spring `@Configuration` or `@SpringBootApplication` class. The moment you explicitly provide a bean definition that would otherwise be provided by Spring Data for Pivotal GemFire using 1 of the annotations, the annotation-based configuration backs away.

In certain cases, you may even need to fall back to Java configuration, as in the `Configurers` case, to handle more complex or conditional configuration logic that is not easily expressed in or cannot be accomplished by using annotations alone. Do not be alarmed. This behavior is to be expected.

For example, another case where you need Java or XML configuration is when configuring Pivotal GemFire WAN components, which currently do not have any annotation configuration support. However, defining and registering WAN components requires only using the `org.springframework.data.gemfire.wan.GatewayReceiverFactoryBean` and `org.springframework.data.gemfire.wan.GatewaySenderFactoryBean` API classes in the Java configuration of your Spring `@Configuration` or `@SpringBootApplication` classes (recommended).

The annotations were not meant to handle every situation. The annotations were meant to help you *get up and running* as *quickly* and as *easily* as possible, especially during development.

We hope you will enjoy these new capabilities!

7.19. Annotation-based Configuration Quick Start

The following sections provide an overview to the SDG annotations in order to get started quickly.

All annotations provide additional configuration attributes along with associated properties to conveniently customize the configuration and behavior of Pivotal GemFire at runtime. However, in general, none of the attributes or associated properties are required to use a particular Pivotal GemFire feature. Simply declare the annotation to enable the feature and you are done. Refer to the individual Javadoc of each annotation for more details.

7.19.1. Configure a `ClientCache` Application

To configure and bootstrap a Pivotal GemFire `ClientCache` application, use the following:

```
@SpringBootApplication
@ClientCacheApplication
public class ClientApplication {

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

See `@ClientCacheApplication` Javadoc.

See Configuring Pivotal GemFire Applications with Spring for more details.

7.19.2. Configure a Peer `Cache` Application

To configure and bootstrap a Pivotal GemFire Peer `Cache` application, use the following:

```
@SpringBootApplication
@PeerCacheApplication
public class ServerApplication {

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

If you would like to enable a `CacheServer` that allows `ClientCache` applications to connect to this server, then simply replace the `@PeerCacheApplication` annotation with the `@CacheServerApplication` annotation. This will start a `CacheServer` running on "localhost", listening on the default `CacheServer` port of `40404`.

See `@CacheServerApplication` Javadoc.

See `@PeerCacheApplication` Javadoc.

See Configuring Pivotal GemFire Applications with Spring for more details.

7.19.3. Configure an Embedded Locator

Annotate your Spring `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableLocator` to start an embedded Locator bound to all NICs listening on the default Locator port, `10334`, as follows:

```
@SpringBootApplication
@CacheServerApplication
@EnableLocator
public class ServerApplication {

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

`@EnableLocator` can only be used with Pivotal GemFire server applications.

See `@EnableLocator` Javadoc.

See Configuring an Embedded Locator for more details.

7.19.4. Configure an Embedded Manager

Annotate your Spring `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableManager` to start an embedded Manager bound to all NICs listening on the default Manager port, `1099`, as follows:

```
@SpringBootApplication
@CacheServerApplication
@EnableManager
public class ServerApplication {

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

`@EnableManager` can only be used with Pivotal GemFire server applications.

See `@EnableManager` Javadoc.

See Configuring an Embedded Manager for more details.

7.19.5. Configure the Embedded HTTP Server

Annotate your Spring `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableHttpService` to start the embedded HTTP server (Jetty) listening on port `7070`, as follows:

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

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

`@EnableHttpService` can only be used with Pivotal GemFire server applications.

See `@EnableHttpService` Javadoc.

See Configuring the Embedded HTTP Server for more details.

7.19.6. Configure the Embedded Memcached Server

Annotate your Spring `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableMemcachedServer` to start the embedded Memcached server (Gemcached) listening on port `11211`, as follows:

```
@SpringBootApplication
@CacheServerApplication
@EnableMemcachedServer
public class ServerApplication {

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

`@EnableMemcachedServer` can only be used with Pivotal GemFire server applications.

See `@EnableMemcachedServer` Javadoc.

See Configuring the Embedded Memcached Server (Gemcached) for more details.

7.19.7. Configure the Embedded Redis Server

Annotate your Spring `@PeerCacheApplication` or `@CacheServerApplication` class with `@EnableRedisServer` to start the embedded Redis server listening on port `6379`, as follows:

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

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

`@EnableRedisServer` can only be used with Pivotal GemFire server applications.

See `@EnableRedisServer` Javadoc.

See Configuring the Embedded Redis Server for more details.

7.19.8. Configure Logging

To configure or adjust Pivotal GemFire logging, annotate your Spring, Pivotal GemFire client or server application class with `@EnableLogging`, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableLogging(logLevel="trace")
public class ClientApplication {

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

Default `log-level` is “config”. Also, this annotation will not adjust log levels in your application, only for Pivotal GemFire.

See `@EnableLogging` Javadoc.

See Configuring Logging for more details.

7.19.9. Configure Statistics

To gather Pivotal GemFire statistics at runtime, annotate your Spring, Pivotal GemFire client or server application class with `@EnableStatistics`, as follows:

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

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

See `@EnableStatistics` Javadoc.

See Configuring Statistics for more details.

7.19.10. Configure PDX

To enable Pivotal GemFire PDX serialization, annotate your Spring, Pivotal GemFire client or server application class with `@EnablePdx`, as follows:

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

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

Pivotal GemFire PDX Serialization is an alternative to Java Serialization with many added benefits. For one, it makes short work of making all of your application domain model types serializable without having to implement `java.io.Serializable`.

By default, SDG configures the `MappingPdxSerializer` to serialize your application domain model types, which does not require any special configuration out-of-the-box in order to properly identify application domain objects that need to be serialized and htne perform the serialization since, the logic in `MappingPdxSerializer` is based on Spring Data's mapping infrastructure. See [mapping.pdx-serialize] for more details.

See `@EnablePdx` Javadoc.

See Configuring PDX for more details.

7.19.11. Configure SSL

To enable Pivotal GemFire SSL, annotate your Spring, Pivotal GemFire client or server application class with `@EnableSsl`, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableSsl(components = SERVER)
public class ClientApplication {

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

Minimally, Pivotal GemFire requires you to specify a keystore & truststore using the appropriate configuration attributes or properties. Both keystore & truststore configuration attributes or properties may refer to the same `KeyStore` file. Additionally, you will need to specify a username and password to access the `KeyStore` file if the file has been secured.

Pivotal GemFire SSL allows you to configure the specific components of the system that require TLS, such as client/server, Locators, Gateways, etc. Optionally, you can specify that all components of Pivotal GemFire use SSL with “ALL”.

See `@EnableSsl` Javadoc.

See Configuring SSL for more details.

7.19.12. Configure Security

To enable Pivotal GemFire security, annotate your Spring, Pivotal GemFire client or server application class with `@EnableSecurity`, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableSecurity
public class ClientApplication {

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

On the server, you must configure access to the auth credentials. You may either implement the Pivotal GemFire `SecurityManager` interface or declare 1 or more Apache Shiro `Realms`. See Configuring Server Security for more details.

On the client, you must configure a username and password. See Configuring Client Security for more details.

See `@EnableSecurity` Javadoc.

See Configuring Security for more details.

7.19.13. Configure Pivotal GemFire Properties

To configure other, low-level Pivotal GemFire properties not covered by the feature-oriented, SDG configuration annotations, annotate your Spring, Pivotal GemFire client or server application class with `@GemFireProperties`, as follows:

```
@SpringBootApplication
@PeerCacheApplication
@EnableGemFireProperties(
    cacheXmlFile = "/path/to/cache.xml",
    conserveSockets = true,
    groups = "GroupOne",
    remoteLocators = "lunchbox[11235],mailbox[10101],skullbox[12480]"
)
public class ServerApplication {
```

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

Some Pivotal GemFire properties are client-side only while others are server-side only. Please review the Pivotal GemFire docs for the appropriate use of each property.

See `@EnableGemFireProperties` Javadoc.

See Configuring Pivotal GemFire Properties for more details.

7.19.14. Configure Caching

To use Pivotal GemFire as a *caching provider* in Spring's *Cache Abstraction*, and have SDG automatically create Pivotal GemFire Regions for the caches required by your application service components, then annotate your Spring, Pivotal GemFire client or server application class with `@EnableGemfireCaching` and `@EnableCachingDefinedRegions`, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableCachingDefinedRegions
@EnableGemfireCaching
public class ClientApplication {

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

Then, simply go on to define the application services that require caching, as follows:

```
@Service
public class BookService {

    @Cacheable("Books")
    public Book findBy(ISBN isbn) {
        ...
    }
}
```

`@EnableCachingDefinedRegions` is optional. That is, you may manually define your Regions if you desire.

See `@EnableCachingDefinedRegions` Javadoc.

See `@EnableGemfireCaching` Javadoc.

See Configuring Spring's Cache Abstraction for more details.

7.19.15. Configure Regions, Indexes, Repositories and Entities for Persistent Applications

To make short work of creating Spring, Pivotal GemFire persistent client or server applications, annotate your application class with `@EnableEntityDefinedRegions`, `@EnableGemfireRepositories` and `@EnableIndexing`, as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableEntityDefinedRegions(basePackageClasses = Book.class)
@EnableGemfireRepositories(basePackageClasses = BookRepository.class)
@EnableIndexing
public class ClientApplication {

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

The `@EnableEntityDefinedRegions` annotation is required when using the `@EnableIndexing` annotation. See Configuring Indexes for more details.

Next, define your entity class and use the `@Region` mapping annotation to specify the Region in which your entity will be stored. Use the `@Indexed` annotation to define Indexes on entity fields used in your application queries, as follows:

```
package example.app.model;

import ...;

@Region("Books")
public class Book {

  @Id
  private ISBN isbn;

  @Indexed;
  private Author author;

  @Indexed
  private LocalDate published;

  @LuceneIndexed
  private String title;

}
```

The `@Region("Books")` entity class annotation is used by the `@EnableEntityDefinedRegions` to determine the Regions required by the application. See Configuring Type-specific Regions and POJO Mapping for more details.

Finally, define your CRUD Repository with simple queries to persist and access `Books`, as follows:

```
package example.app.repo;

import ...;

public interface BookRepository extends CrudRepository {
```

```
  List<Book> findByAuthorOrderByPublishedDesc(Author author);

}
```

See Spring Data for Pivotal GemFire Repositories for more details.

See `@EnableEntityDefinedRegions` Javadoc.

See `@EnableGemfireRepositories` Javadoc.

See `@EnableIndexing` Javadoc.

See `@Region` Javadoc.

See `@Indexed` Javadoc.

See `@LuceneIndexed` Javadoc.

See Configuring Regions for more details.

See Spring Data for Pivotal GemFire Repositories for more details.

7.19.16. Configure Client Regions from Cluster-defined Regions

Alternatively, you can define client [*PROXY] Regions from Regions already defined in the cluster using `@EnableClusterDefinedRegions` , as follows:

```
@SpringBootApplication
@ClientCacheApplication
@EnableClusterDefinedRegions
@EnableGemfireRepositories
public class ClientApplication {

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

  ...
}
```

See Configured Cluster-defined Regions for more details.

7.19.17. Configure Functions

Pivotal GemFire Functions are useful in distributed compute scenarios where a potentially expensive computation requiring data can be performed in parallel across the nodes in the cluster. In this case, it is more efficient to bring the logic to where the data is located (stored) rather than requesting and fetching the data to be processed by the computation.

Use the `@EnableGemfireFunctions` along with the `@GemfireFunction` annotation to enable Pivotal GemFire Functions definitions implemented as methods on POJOs, as follows:

```
@PeerCacheApplication
@EnableGemfireFunctions
class ServerApplication {

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

  @GemfireFunction
  Integer computeLoyaltyPoints(Customer customer) {
    ...
  }
}
```

Use the `@EnableGemfireFunctionExecutions` along with 1 of the Function calling annotations: `@OnMember`, `@OnMembers`, `@OnRegion`, `@OnServer` and `@OnServers`.

```
@ClientCacheApplication
@EnableGemfireFunctionExecutions(basePackageClasses = CustomerRewardsFunction.class)
class ClientApplication {

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

@OnRegion("Customers")
interface CustomerRewardsFunctions {

  Integer computeLoyaltyPoints(Customer customer);

}
```

See `@EnableGemfireFunctions` Javadoc.

See `@GemfireFunction` Javadoc.

See `@EnableGemfireFunctionExecutions` Javadoc.

See `@OnMember` Javadoc, `@OnMembers` Javadoc, `@OnRegion` Javadoc, `@OnServer` Javadoc, and `@OnServers` Javadoc.

See Annotation Support for Function Execution for more details.

7.19.18. Configure Continuous Query

Real-time, event stream processing is becoming an increasingly important task for data-intensive applications, primarily in order to respond to user requests in a timely manner. Pivotal GemFire Continuous Query (CQ) will help you achieve this rather complex task quite easily.

Enable CQ by annotating your application class with `@EnableContinuousQueries` and define your CQs along with the associated event handlers, as follows:

```
@ClientCacheApplication
@EnableContinuousQueries
class ClientApplication {

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

Then, define your CQs by annotating the associated handler method with `@ContinousQuery`, as follows:

```
@Service
class CustomerService {

  @ContinuousQuery(name = "CustomerQuery", query = "SELECT * FROM /Customers c WHERE ...")
  public void process(CqEvent event) {
    ...
  }
}
```

Anytime an event occurs changing the `Customer` data to match the predicate in your CQ query, the `process` method will be called.

Pivotal GemFire CQ is a client-side feature only.

See `@EnableContinuousQueries` Javadoc.

See `@ContinuousQuery` Javadoc.

See Continuous Query (CQ) and Configuring Continuous Queries for more details.

---

8. Working with Pivotal GemFire APIs

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

8.1. GemfireTemplate

As with many other high-level abstractions provided by Spring, Spring Data for Pivotal GemFire provides a **template** to simplify Pivotal GemFire data access operations. The class provides several methods containing common Region operations, but also provides the capability to **execute** code against native Pivotal GemFire APIs without having to deal with Pivotal GemFire checked exceptions by using a `GemfireCallback`.

The template class requires a Pivotal GemFire `Region`, and once configured, is thread-safe and is reusable 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 Pivotal GemFire `Region` without having to deal with checked exceptions, threading or resource management concerns:

```
template.execute(new GemfireCallback<Iterable<String>>() {

    public Iterable<String> doInGemfire(Region 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 Pivotal GemFire query language, a developer can use the `find` and `findUnique` methods, which, compared 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.

8.2. 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 Pivotal GemFire 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.

8.3. 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 Pivotal GemFire, Spring Data for Pivotal GemFire 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"/>
```

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

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

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

8.4. Global, JTA Transaction Management

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

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

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

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

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

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

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

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

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

Peer Cache XML:

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

Peer Cache JavaConfig:

```
@Bean
CacheFactoryBean 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
ClientCacheFactoryBean gemfireCache() {

  ClientCacheFactoryBean gemfireCache = new ClientCacheFactoryBean();

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

  return gemfireCache;
}
```

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 Pivotal GemFire's documentation and let *Spring Data Geode* work its magic. All you need do is annotate your Spring `@Configuration` class with Spring Data for Pivotal GemFire's **new** `@EnableGemFireAsLastResource` annotation and a combination of Spring's Transaction Management infrastructure and Spring Data for Pivotal GemFire's `@EnableGemFireAsLastResource` configuration does the trick.

The configuration looks like this…

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

  ...
}
```

The only requirements are…

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

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

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

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

   JtaTransactionManager transactionManager = new JtaTransactionManager();

   transactionManager.setUserTransaction(userTransaction);

   return transactionManager;
}
```

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

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

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

Specifically, the correct sequence of events are…

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

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

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

```
@Service
class MyTransactionalService ... {

  @Transactional
  public <Return-Type> someTransactionalServiceMethod() {
    // 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.someTransactionalServiceMethod()` is called).

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

#3 of course is the responsibility of your application.

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

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

2017-Jun-22 11:11:37 TRACE GemFireAsLastResourceConnectionAcquiringAspect - Acquiring {data-store-name} Connection
from {data-store-name} 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 {data-store-name} Connection @ [Reference
[...]]
```

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

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

8.5. Continuous Query (CQ)

A powerful functionality offered by Pivotal GemFire is Continuous Query (or CQ).

In short, CQ allows a developer to create and register an OQL query, and then automatically be notified when new data that gets added to Pivotal GemFire matches the query predicate. Spring Data for Pivotal GemFire provides dedicated support for CQs through the `org.springframework.data.gemfire.listener` package and its **listener container**; very similar in functionality and naming to the JMS integration in the *Spring Framework*; in fact, users familiar with the JMS support in Spring, should feel right at home.

Basically Spring Data for Pivotal GemFire 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 for Pivotal GemFire 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 Pivotal GemFire.

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

8.5.1. Continuous Query Listener Container

Spring Data for Pivotal GemFire 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 Pivotal GemFire 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.

8.5.2. The `ContinuousQueryListener` and `ContinuousQueryListenerAdapter`

The `ContinuousQueryListenerAdapter` class is the final component in Spring Data for Pivotal GemFire 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 Pivotal GemFire'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** Pivotal GemFire dependencies at all. It truly is a POJO that we can and will make into an EDP via the following configuration.

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>
```

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 for Pivotal GemFire namespace to declare the event listener container and automatically register the listeners. The full blown, **beans** definition is displayed below:

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

<!-- and this is the event listener container... -->
<bean id="gemfireListenerContainer" class="org.springframework.data.gemfire.listener.ContinuousQueryListenerContainer">
    <property name="cache" ref="gemfireCache"/>
    <property name="queryListeners">
      <!-- set of 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 Pivotal GemFire event and the required method argument(s) transparently. Any exception caused by the method invocation is caught and handled by the container (by default, being logged).

8.6. Wiring `Declarable` Components

Pivotal GemFire 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 Pivotal GemFire. Using native Pivotal GemFire configuration, each user type declared through XML must implement the `Declarable` interface, which allows arbitrary parameters to be passed to the declared class through a `Properties` instance.

In this section, we describe how you can configure these pluggable components 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 Pivotal GemFire 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 for Pivotal GemFire offers a base class (`WiringDeclarableSupport`) that allows Pivotal GemFire 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 Pivotal GemFire release there is no concept of an **object factory** and the types declared are instantiated and used as is. In other words, there is no easy way to manage object creation outside Pivotal GemFire.

8.6.1. 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 Pivotal GemFire. For cases where the bean name uses a different convention, one can pass in the `bean-name` parameter in the Pivotal GemFire configuration:

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

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

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

8.6.2. 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.

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.

8.7. Support for the Spring Cache Abstraction

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

To use Pivotal GemFire 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/cache http://www.springframework.org/schema/cache/spring-cache.xsd
    http://www.springframework.org/schema/gemfire http://www.springframework.org/schema/gemfire/spring-gemfire.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>
```

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 Pivotal GemFire 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/cache http://www.springframework.org/schema/cache/spring-cache.xsd
    http://www.springframework.org/schema/gemfire http://www.springframework.org/schema/gemfire/spring-gemfire.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.

---

9. Working with Pivotal GemFire Serialization

To improve overall performance of the Pivotal GemFire In-memory Data Grid, Pivotal GemFire supports a dedicated serialization protocol, called PDX, that is both faster and offers more compact results over standard Java serialization in addition to working transparently across various language platforms (Java, C++, and .NET).

See PDX Serialization Features and PDX Serialization Internals for more details.

This chapter discusses the various ways in which Spring Data for Pivotal GemFire simplifies and improves Pivotal GemFire's custom serialization in Java.

9.1. 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 or machine. For such cases, Spring Data for Pivotal GemFire offers a special `Instantiator` that performs wiring for each new instance created by Pivotal GemFire during deserialization.

Through such a mechanism, you 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 similarly to `WiringDeclarableSupport`, trying to first locate a bean definition as a wiring template and otherwise falling back to auto-wiring.

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

To use the SDG `Instantiator`, declare it as a bean, as the following example shows:

```
<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 has been initialized, the `Instantiator`, by default, registers itself with the Pivotal GemFire serialization system and performs wiring on all instances of `SomeDataSerializableClass` created by Pivotal GemFire during deserialization.

9.2. Auto-generating Custom `Instantiators`

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

```
<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 preceding definition automatically generates two `Instantiators` for two classes (`CustomTypeA` and `CustomTypeB`) and registers them with Pivotal GemFire under user ID `1025` and `1026`. The two `Instantiators` avoid the use of reflection and create the instances directly through Java code.

10. POJO Mapping

This section covers:

- Entity Mapping
- Repository Mapping
- MappingPdxSerializer

10.1. Object Mapping Fundamentals

This section covers the fundamentals of Spring Data object mapping, object creation, field and property access, mutability and immutability. Note, that this section only applies to Spring Data modules that do not use the object mapping of the underlying data store (like JPA). Also be sure to consult the store-specific sections for store-specific object mapping, like indexes, customizing column or field names or the like.

Core responsibility of the Spring Data object mapping is to create instances of domain objects and map the store-native data structures onto those. This means we need two fundamental steps:

1. Instance creation by using one of the constructors exposed.
2. Instance population to materialize all exposed properties.

10.1.1. Object creation

Spring Data automatically tries to detect a persistent entity's constructor to be used to materialize objects of that type. The resolution algorithm works as follows:

1. If there's a no-argument constructor, it will be used. Other constructors will be ignored.
2. If there's a single constructor taking arguments, it will be used.
3. If there are multiple constructors taking arguments, the one to be used by Spring Data will have to be annotated with `@PersistenceConstructor`.

The value resolution assumes constructor argument names to match the property names of the entity, i.e. the resolution will be performed as if the property was to be populated, including all customizations in mapping (different datastore column or field name etc.). This also requires either parameter names information available in the class file or an `@ConstructorProperties` annotation being present on the constructor.

The value resolution can be customized by using Spring Framework's `@Value` value annotation using a store-specific SpEL expression. Please consult the section on store specific mappings for further details.

Object creation internals

To avoid the overhead of reflection, Spring Data object creation uses a factory class generated at runtime by default, which will call the domain classes constructor directly. I.e. for this example type:

```
class Person {
  Person(String firstname, String lastname) { … }
```

```
}
```

we will create a factory class semantically equivalent to this one at runtime:

```
class PersonObjectInstantiator implements ObjectInstantiator {

  Object newInstance(Object... args) {
    return new Person((String) args[0], (String) args[1]);
  }
}
```

This gives us a roundabout 10% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- it must not be a private class
- it must not be a non-static inner class
- it must not be a CGLib proxy class
- the constructor to be used by Spring Data must not be private

If any of these criteria match, Spring Data will fall back to entity instantiation via reflection.

10.1.2. Property population

Once an instance of the entity has been created, Spring Data populates all remaining persistent properties of that class. Unless already populated by the entity's constructor (i.e. consumed through its constructor argument list), the identifier property will be populated first to allow the resolution of cyclic object references. After that, all non-transient properties that have not already been populated by the constructor are set on the entity instance. For that we use the following algorithm:

1. If the property is immutable but exposes a wither method (see below), we use the wither to create a new entity instance with the new property value.
2. If property access (i.e. access through getters and setters) is defined, we're invoking the setter method.
3. By default, we set the field value directly.

Property population internals

Similarly to our optimizations in object construction we also use Spring Data runtime generated accessor classes to interact with the entity instance.

```
class Person {

  private final Long id;
  private String firstname;
  private @AccessType(Type.PROPERTY) String lastname;

  Person() {
    this.id = null;
  }

  Person(Long id, String firstname, String lastname) {
    // Field assignments
  }
```

```
  Person withId(Long id) {
    return new Person(id, this.firstname, this.lastame);
  }

  void setLastname(String lastname) {
    this.lastname = lastname;
  }
}
```

Example 1. A generated Property Accessor

```
class PersonPropertyAccessor implements PersistentPropertyAccessor {

  private static final MethodHandle firstname;              2

  private Person person;                                    1

  public void setProperty(PersistentProperty property, Object value) {

    String name = property.getName();

    if ("firstname".equals(name)) {
      firstname.invoke(person, (String) value);             2
    } else if ("id".equals(name)) {
      this.person = person.withId((Long) value);            3
    } else if ("lastname".equals(name)) {
      this.person.setLastname((String) value);              4
    }
  }
}
```

1. PropertyAccessor's hold a mutable instance of the underlying object. This is, to enable mutations of otherwise immutable properties.
2. By default, Spring Data uses field-access to read and write property values. As per visibility rules of `private` fields, `MethodHandles` are used to interact with fields.
3. The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. Calling `withId(…)` creates a new `Person` object. All subsequent mutations will take place in the new instance leaving the previous untouched.
4. Using property-access allows direct method invocations without using `MethodHandles`.

This gives us a roundabout 25% performance boost over reflection. For the domain class to be eligible for such optimization, it needs to adhere to a set of constraints:

- Types must not reside in the default or under the `java` package.
- Types and their constructors must be `public`
- Types that are inner classes must be `static`.
- The used Java Runtime must allow for declaring classes in the originating `ClassLoader`. Java 9 and newer impose certain limitations.

By default, Spring Data attempts to use generated property accessors and falls back to reflection-based ones if a limitation is detected.

Let's have a look at the following entity:

Example 2. A sample entity

```
class Person {

  private final @Id Long id;                                                  1
  private final String firstname, lastname;                                   2
  private final LocalDate birthday;
  private final int age;  3

  private String comment;                                                     4
  private @AccessType(Type.PROPERTY) String remarks;                          5

  static Person of(String firstname, String lastname, LocalDate birthday) {  6

    return new Person(null, firstname, lastname, birthday,
      Period.between(birthday, LocalDate.now()).getYears());
  }

  Person(Long id, String firstname, String lastname, LocalDate birthday, int age) {  6

    this.id = id;
    this.firstname = firstname;
    this.lastname = lastname;
    this.birthday = birthday;
    this.age = age;
  }

  Person withId(Long id) {                                                    1
    return new Person(id, this.firstname, this.lastname, this.birthday);
  }

  void setRemarks(String remarks) {                                           5
    this.remarks = remarks;
  }
}
```

1 The identifier property is final but set to `null` in the constructor. The class exposes a `withId(…)` method that's used to set the identifier, e.g. when an instance is inserted into the datastore and an identifier has been generated. The original `Person` instance stays unchanged as a new one is created. The same pattern is usually applied for other properties that are store managed but might have to be changed for persistence operations.

2 The `firstname` and `lastname` properties are ordinary immutable properties potentially exposed through getters.

3 The `age` property is an immutable but derived one from the `birthday` property. With the design shown, the database value will trump the defaulting as Spring Data uses the only declared constructor. Even if the intent is that the calculation should be preferred, it's important that this constructor also takes `age` as parameter (to potentially ignore it) as otherwise the property population step will attempt to set the age field and fail due to it being immutable and no wither being present.

4 The `comment` property is mutable is populated by setting its field directly.

5 The `remarks` properties are mutable and populated by setting the `comment` field directly or by invoking the setter method for

6 The class exposes a factory method and a constructor for object creation. The core idea here is to use factory methods instead of additional constructors to avoid the need for constructor disambiguation through `@PersistenceConstructor`. Instead, defaulting of properties is handled within the factory method.

10.1.3. General recommendations

- *Try to stick to immutable objects* — Immutable objects are straightforward to create as materializing an object is then a matter of calling its constructor only. Also, this avoids your domain objects to be littered with setter methods that allow

client code to manipulate the objects state. If you need those, prefer to make them package protected so that they can only be invoked by a limited amount of co-located types. Constructor-only materialization is up to 30% faster than properties population.

- *Provide an all-args constructor* — Even if you cannot or don't want to model your entities as immutable values, there's still value in providing a constructor that takes all properties of the entity as arguments, including the mutable ones, as this allows the object mapping to skip the property population for optimal performance.
- *Use factory methods instead of overloaded constructors to avoid* `@PersistenceConstructor` — With an all-argument constructor needed for optimal performance, we usually want to expose more application use case specific constructors that omit things like auto-generated identifiers etc. It's an established pattern to rather use static factory methods to expose these variants of the all-args constructor.
- *Make sure you adhere to the constraints that allow the generated instantiator and property accessor classes to be used* —
- *For identifiers to be generated, still use a final field in combination with a wither method* —
- *Use Lombok to avoid boilerplate code* — As persistence operations usually require a constructor taking all arguments, their declaration becomes a tedious repetition of boilerplate parameter to field assignments that can best be avoided by using Lombok's `@AllArgsConstructor`.

10.1.4. Kotlin support

Spring Data adapts specifics of Kotlin to allow object creation and mutation.

Kotlin object creation

Kotlin classes are supported to be instantiated , all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person`:

```
data class Person(val id: String, val name: String)
```

The class above compiles to a typical class with an explicit constructor. We can customize this class by adding another constructor and annotate it with `@PersistenceConstructor` to indicate a constructor preference:

```
data class Person(var id: String, val name: String) {

    @PersistenceConstructor
    constructor(id: String) : this(id, "unknown")
}
```

Kotlin supports parameter optionality by allowing default values to be used if a parameter is not provided. When Spring Data detects a constructor with parameter defaulting, then it leaves these parameters absent if the data store does not provide a value (or simply returns `null`) so Kotlin can apply parameter defaulting. Consider the following class that applies parameter defaulting for `name`

```
data class Person(var id: String, val name: String = "unknown")
```

Every time the `name` parameter is either not part of the result or its value is `null`, then the `name` defaults to `unknown`.

Property population of Kotlin data classes

In Kotlin, all classes are immutable by default and require explicit property declarations to define mutable properties. Consider the following `data class Person`:

```
data class Person(val id: String, val name: String)
```

This class is effectively immutable. It allows to create new instances as Kotlin generates a `copy(…)` method that creates new object instances copying all property values from the existing object and applying property values provided as arguments to the method.

10.2. Entity Mapping

Spring Data for Pivotal GemFire provides support to map entities that are stored in a Region. The mapping metadata is defined by using annotations on application domain classes, as the following example shows:

Example 3. Mapping a domain class to a Pivotal GemFire Region

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

  @Id Long id;

  String firstname;
  String lastname;

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

  …
}
```

The `@Region` annotation 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 should 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, as the following example shows:

```
@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 Pivotal GemFire Region, as defined with the Spring Data for Pivotal GemFire XML namespace by using the `id` or `name` attributes of the `<*-region>` element.

10.2.1. Entity Mapping by Region Type

In addition to the `@Region` annotation, Spring Data for Pivotal GemFire also recognizes type-specific Region 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 for Pivotal GemFire's annotation configuration model. When combined with the `@EnableEntityDefinedRegions` configuration annotation on a Spring `@Configuration` annotated class, it is possible to generate Regions in the local cache, whether the application is a client or peer.

These annotations let you be more specific about what type of Region your application entity class should be mapped to and also has an impact on the data management policies of the Region (for example, partition — also known as sharding — versus replicating data).

Using these type-specific Region mapping annotations with the SDG annotation configuration model saves you from having to explicitly define these Regions in configuration.

10.3. Repository Mapping

As an alternative to specifying the Region in which the entity is stored by using the `@Region` annotation on the entity class, you can also specify the `@Region` annotation on the entity's `Repository` interface. See Spring Data for Pivotal GemFire Repositories for more details.

However, suppose you want to store a `Person` record in multiple Pivotal GemFire Regions (for example, `People` and `Customers`). Then you can define your corresponding `Repository` interface extensions as follows:

```
@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 Pivotal GemFire Regions, as the following example shows:

```
@Service
class CustomerService {

  CustomerRepository customerRepo;

  PersonRepository personRepo;

  Customer update(Customer customer) {
    customerRepo.save(customer);
    personRepo.save(customer);
    return customer;
  }
```

You can even wrap the `update` service method in a Spring managed transaction, either as a local cache transaction or a global transaction.

10.4. MappingPdxSerializer

Spring Data for Pivotal GemFire provides a custom `PdxSerializer` implementation, called `MappingPdxSerializer`, that uses Spring Data mapping metadata to customize entity serialization.

The serializer also lets you customize entity instantiation by using the Spring Data `EntityInstantiator` abstraction. By default, the serializer use the `ReflectionEntityInstantiator`, which uses the persistence constructor of the mapped entity. The persistence constructor is either the default constructor, a singly declared constructor, or a constructor explicitly annotated with `@PersistenceConstructor`.

To provide arguments for constructor parameters, the serializer reads fields with the named constructor parameter, explicitly identified by using Spring's `@Value` annotation, from the supplied `PdxReader`, as shown in the following example:

Example 4. Using `@Value` on entity constructor parameters

```
public class Person {

  public Person(@Value("#root.thing") String firstName, @Value("bean") String lastName) {
    …
  }
}
```

An entity class annotated in this way has the "thing" field read from the `PdxReader` and passed as the argument value for the constructor parameter, `firstname`. The value for `lastName` is a Spring bean with the name "bean".

In addition to the custom instantiation logic and strategy provided by `EntityInstantiators`, the `MappingPdxSerializer` also provides capabilities well beyond Pivotal GemFire's own `ReflectionBasedAutoSerializer`.

While Pivotal GemFire's `ReflectionBasedAutoSerializer` conveniently uses Java Reflection to populate entities and uses regular expressions to identify types that should be handled (serialized and deserialized) by the serializer, it cannot, unlike `MappingPdxSerializer`, perform the following:

- Register custom `PdxSerializer` objects per entity field or property names and types.
- Conveniently identifies ID properties.
- Automatically handles read-only properties.
- Automatically handles transient properties.
- Allows more robust type filtering in a `null` and type-safe manner (for example, not limited to only expressing types with regex).

We now explore each feature of the `MappingPdxSerializer` in a bit more detail.

10.4.1. Custom PdxSerializer Registration

The `MappingPdxSerializer` gives you the ability to register custom `PdxSerializers` based on an entity's field or property names and types.

For example, suppose you have defined an entity type modeling a `User` as follows:

```
package example.app.security.auth.model;

public class User {

  private String name;

  private Password password;

  ...
}
```

While the user's name probably does not require any special logic to serialize the value, serializing the password on the other hand might require additional logic to handle the sensitive nature of the field or property.

Perhaps you want to protect the password when sending the value over the network, between a client and a server, beyond TLS alone, and you only want to store the salted hash. When using the `MappingPdxSerializer`, you can register a custom `PdxSerializer` to handle the user's password, as follows:

Example 5. Registering custom `PdxSerializers` by POJO field/property type

```
Map<?, PdxSerializer> customPdxSerializers = new HashMap<>();

customPdxSerializers.put(Password.class, new SaltedHashPasswordPdxSerializer());

mappingPdxSerializer.setCustomPdxSerializers(customPdxSerializers);
```

After registering the application-defined `SaltedHashPasswordPdxSerializer` instance with the `Password` application domain model type, the `MappingPdxSerializer` will then consult the custom `PdxSerializer` to serialize and deserialize all `Password` objects regardless of the containing object (for example, `User`).

However, suppose you want to customize the serialization of `Passwords` only on `User` objects. To do so, you can register the custom `PdxSerializer` for the `User` type by specifying the fully qualified name of the `Class's` field or property, as the following example shows:

Example 6. Registering custom `PdxSerializers` by POJO field/property name

```
Map<?, PdxSerializer> customPdxSerializers = new HashMap<>();

customPdxSerializers.put("example.app.security.auth.model.User.password", new SaltedHashPasswordPdxSerializer());

mappingPdxSerializer.setCustomPdxSerializers(customPdxSerializers);
```

Notice the use of the fully-qualified field or property name (that is `example.app.security.auth.model.User.password`) as the custom `PdxSerializer` registration key.

You could construct the registration key by using a more logical code snippet, such as the following: `User.class.getName().concat(".password");`. We recommended this over the example shown earlier. The preceding example tried to be as explicit as possible about the semantics of registration.

10.4.2. Mapping ID Properties

Like Pivotal GemFire's `ReflectionBasedAutoSerializer`, SDG's `MappingPdxSerializer` is also able to determine the identifier of the entity. However, `MappingPdxSerializer` does so by using Spring Data's mapping metadata, specifically by finding the entity property designated as the identifier using Spring Data's `@Id` annotation. Alternatively, any field or property named "id", not explicitly annotated with `@Id`, is also designated as the entity's identifier.

For example:

```
class Customer {

  @Id
  Long id;

  ...
}
```

In this case, the `Customer id` field is marked as the identifier field in the PDX type metadata by using `PdxWriter.markIdentifierField(:String)` when the `PdxSerializer.toData(..)` method is called during serialization.

10.4.3. Mapping Read-only Properties

What happens when your entity defines a read-only property?

First, it is important to understand what a "read-only" property is. If you define a POJO by following the JavaBeans specification (as Spring does), you might define a POJO with a read-only property, as follows:

```
package example;

class ApplicationDomainType {

  private AnotherType readOnly;

  public AnotherType getReadOnly() [
    this.readOnly;
  }

  ...
}
```

The `readOnly` property is read-only because it does not provide a setter method. It only has a getter method. In this case, the `readOnly` property (not to be confused with the `readOnly DomainType` field) is considered read-only.

As a result, the `MappingPdxSerializer` will not try to set a value for this property when populating an instance of `ApplicationDomainType` in the `PdxSerializer.fromData(:Class<ApplicationDomainType>, :PdxReader)` method during deserialization, particularly if a value is present in the PDX serialized bytes.

This is useful in situations where you might be returning a view or projection of some entity type and you only want to set state that is writable. Perhaps the view or projection of the entity is based on authorization or some other criteria. The point is, you can leverage this feature as is appropriate for your application's use cases and requirements. If you want the field or property to always be written, simply define a setter method.

10.4.4. Mapping Transient Properties

Likewise, what happens when your entity defines `transient` properties?

You would expect the `transient` fields or properties of your entity not to be serialized to PDX when serializing the entity. That is exactly what happens, unlike Pivotal GemFire's own `ReflectionBasedAutoSerializer`, which serializes everything accessible from the object through Java Reflection.

The `MappingPdxSerializer` will not serialize any fields or properties that are qualified as being transient, either by using Java's own `transient` keyword (in the case of class instance fields) or by using the `@Transient` Spring Data annotation on either fields or properties.

For example, you might define an entity with transient fields and properties as follows:

```
package example;

class Process {

  private transient int id;

  private File workingDirectory;

  private String name;

  private Type type;

  @Transient
  public String getHostname() {
    ...
  }

  ...
}
```

Neither the `Process id` field nor the readable `hostname` property are written to PDX.

10.4.5. Filtering by Class Type

Similar to Pivotal GemFire's `ReflectionBasedAutoSerializer`, SDG's `MappingPdxSerializer` lets you filter the types of objects that are serialized and deserialized.

However, unlike Pivotal GemFire's `ReflectionBasedAutoSerializer`, which uses complex regular expressions to express which types the serializer handles, SDG's `MappingPdxSerializer` uses the much more robust `java.util.function.Predicate` interface and API to express type-matching criteria.

If you like to use regular expressions, you can implement a `Predicate` using Java's regular expression support.

The nice part about Java's `Predicate` interface is that you can compose `Predicates` by using convenient and appropriate API methods, including: `and(:Predicate)`, `or(:Predicate)`, and `negate()`.

The following example shows the `Predicate` API in action:

```
Predicate<Class<?>> customerTypes =
  type -> Customer.class.getPackage().getName().startsWith(type.getName()); // Include all types in the same package as
`Customer`

Predicate includedTypes = customerTypes
  .or(type -> User.class.isAssignble(type)); // Additionally, include User sub-types (e.g. Admin, Guest, etc)

mappingPdxSerializer.setIncludeTypeFilters(includedTypes);

mappingPdxSerializer.setExcludeTypeFilters(
  type -> !Reference.class.getPackage(type.getPackage()); // Exclude Reference types
```

Any `Class` object passed to your `Predicate` is guaranteed not to be `null`.

SDG's `MappingPdxSerializer` includes support for both include and exclude class type filters.

Exclude Type Filtering

By default, SDG's `MappingPdxSerializer` registers pre-defined `Predicates` that filter, or exclude types from the folliowing packages:

- `java.*`
- `com.gemstone.gemfire.*`
- `org.apache.geode.*`
- `org.springframework.*`

In addition, the `MappingPdxSerializer` filters `null` objects when calling `PdxSerializer.toData(:Object, :PdxWriter)` and `null` class types when calling `PdxSerializer.fromData(:Class<?>, :PdxReader)` methods.

It is very easy to add exclusions for other class types, or an entire package of types, by simply defining a `Predicate` and adding it to the `MappingPdxSerializer` as shown earlier.

The `MappingPdxSerializer.setExcludeTypeFilters(:Predicate<Class<?>>)` method is additive, meaning it composes your application-defined type filters with the existing, pre-defined type filter `Predicates` indicated above using the `Predicate.and(:Predicate<Class<?>>)` method.

However, what if you want to include a class type (for example, `java.security Principal`) implicitly excluded by the exclude type filters? See Include Type Filtering.

Include Type Filtering

If you want to include a class type explicitly, or override a class type filter that implicitly excludes a class type required by your application (for example, `java.security.Principal`, which is excluded by default with the `java.*` package exclude type filter on `MappingPdxSerializer`), then just define the appropriate `Predicate` and add it to the serializer using `MappingPdxSerializer.setIncludeTypeFilters(:Predicate<Class<?>>)` method, as follows:

```
Predicate<Class<?>> principalTypeFilter =
  type -> java.security.Principal.class.isAssignableFrom(type);

mappingPdxSerializer.setIncludeTypeFilters(principalTypeFilters);
```

Again, the `MappingPdxSerializer.setIncludeTypeFilters(:Predicate<Class<?>>)` method, like `setExcludeTypeFilters(:Predicate<Class<?>>)`, is additive and therefore composes any passed type filter using `Predicate.or(:Predicate<Class<?>>)`. This means you may call `setIncludeTypeFilters(:Predicate<Class<?>>)` as many time as necessary.

When include type filters are present, then the `MappingPdxSerializer` makes a decision of whether to de/serialize an instance of a class type when the class type is either not implicitly excluded OR when the class type is explicitly included, whichever returns true. Then, an instance of the class type will be serialized or deserialized appropriately.

For example, when a type filter of `Predicate<Class<Principal>>` is explicitly registered as shown previously, it cancels out the implicit exclude type filter on `java.*` package types.

11. Spring Data for Pivotal GemFire Repositories

Spring Data for Pivotal GemFire provides support for using the Spring Data Repository abstraction to easily persist entities into Pivotal GemFire along with executing queries. A general introduction to the Repository programming model is provided here.

11.1. Spring XML Configuration

To bootstrap Spring Data Repositories, use the `<repositories/>` element from the Spring Data for Pivotal GemFire Data namespace, as the following example shows:

Example 7. Bootstrap Spring Data for Pivotal GemFire Repositories in XML

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

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

</beans>
```

The preceding configuration snippet looks for interfaces below the configured base package and creates Repository instances for those interfaces backed by a `SimpleGemFireRepository`.

The bootstrap process fails unless you have your application domain classes correctly mapped to configured Regions.

11.2. Spring Java-based Configuration

Alternatively, many developers prefer to use Spring's Java-based container configuration.

Using this approach, you can bootstrap Spring Data Repositories by using the SDG `@EnableGemfireRepositories` annotation, as the following example shows:

Example 8. Bootstrap Spring Data for Pivotal GemFire Repositories with `@EnableGemfireRepositories`

```
@SpringBootApplication
@EnableGemfireRepositories(basePackages = "com.example.acme.repository")
class SpringDataApplication {
  ...
}
```

Rather than use the `basePackages` attribute, you may prefer to use the type-safe `basePackageClasses` attribute instead. The `basePackageClasses` lets you specify the package that contains all your application Repository classes by specifying only one of your application Repository interface types. Consider creating a special no-op marker class or interface in each package that serves no purpose other than to identify the location of application Repositories referenced by this attribute.

In addition to the `basePackages and basePackageClasses` 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 lets you specify the location of named OQL queries, which reside in a Java `Properties` file, by 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 one or more custom repository implementations. This feature is commonly used to extend the Spring Data Repository infrastructure to implement a feature not provided by the data store (for example, SDG).

One example of where custom repository implementations are needed with Pivotal GemFire is when performing joins. Joins are not supported by SDG Repositories. With a Pivotal GemFire `PARTITION` Region, the join must be performed on collocated `PARTITION` Regions, since Pivotal GemFire does not support "distributed" joins. In addition, the Equi-Join OQL Query must be performed inside a Pivotal GemFire Function. See here for more details on Pivotal GemFire *Equi-Join Queries*.

Many other aspects of the SDG's Repository infrastructure extension may be customized as well. See the `@EnableGemfireRepositories` Javadoc for more details on all configuration settings.

11.3. Executing OQL Queries

Spring Data for Pivotal GemFire Repositories enable the definition of query methods to easily execute Pivotal GemFire OQL queries against the Region the managed entity maps to, as the following example shows:

Example 9. 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 in the preceding example causes 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 returns all entities found, whereas the first query method expects a single result to be found.

If the supported keywords are not sufficient to declare and express your OQL query, or the method name becomes too verbose, then you can annotate the query methods with `@Query` as shown on the third and fourth methods.

The following table gives brief samples of the supported keywords that you can use in query methods:

Table 4. Supported keywords for query methods

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

11.4. OQL Query Extensions Using Annotations

Many query languages, such as Pivotal GemFire'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 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 convenient and powerful abstraction.

To support Pivotal GemFire's OQL Query language extensions and preserve portability across different data stores, Spring Data for Pivotal GemFire adds support for OQL Query extensions by using Java annotations. These annotations are

ignored by other Spring Data Repository implementations (such as Spring Data JPA or Spring Data Redis) that do not have similar query language features.

For instance, many data stores most likely do not implement Pivotal GemFire's OQL `IMPORT` keyword. Implementing `IMPORT` as an annotation (that is, `@Import`) rather than as part of the query method signature (specifically, the method 'name') does not interfere with the parsing infrastructure when evaluating the query method name to construct another data store language appropriate query.

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

Table 5. Supported Pivotal GemFire 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 Pivotal GemFire Region along with a `CustomerRepository` and a query method to lookup `Customers` by last name, as follows:

Example 10. 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);

  ...
}
```

The preceding example results 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 for Pivotal GemFire's Repository extension 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`, as follows:

Example 11. 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);

}
```

The preceding 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
```

The `@Limit(10)` annotation does not override the `LIMIT` explicitly defined in the raw query. Also, the `@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.

The `ReputationIdx` index is probably not the most sensible index, given the number of customers who may possibly have the same value for their reputation, which reduces the effectiveness of the index. Please choose indexes and other optimizations wisely, as an improper or poorly chosen index can have the opposite effect on your performance because of the overhead in maintaining the index. The `ReputationIdx` was used only to serve the purpose of the example.

11.5. Query Post Processing

Thanks to using the Spring Data Repository abstraction, the 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 for Pivotal GemFire includes the `o.s.d.gemfire.repository.query.QueryPostProcessor` functional interface. The interface is loosely defined as follows:

Example 12. 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 that let you compose instances of `QueryPostProcessor` similar to how java.util.function.Function.andThen(:Function) and java.util.function.Function.compose(:Function) work.

Additionally, the `QueryPostProcessor` interface implements the `org.springframework.core.Ordered` interface, which is useful when multiple `QueryPostProcessors` are declared and registered in the Spring container 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 `T` extends the Spring Data Commons marker interface, `org.springframework.data.repository.Repository`. We discuss this further later in this section. All `QUERY` type parameter arguments in Spring Data for Pivotal GemFire's case are of type `java.lang.String`.

It is useful to define the query as type `QUERY`, since this `QueryPostProcessor` interface may be ported to Spring Data Commons and therefore must handle all forms of queries by different data stores (such as JPA, MongoDB, or Redis).

You can 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, you might want to log all queries from all application Repository interface definitions. You could do so by using the following `QueryPostProcessor` implementation:

Example 13. 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, logs 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, a `CustomerRepository`, as the following example shows:

Example 14. CustomerRepository

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

  Customer findByAccountNumber(String accountNumber);

  List<Customer> findByLastNameLike(String lastName);

}
```

Then you could have typed the `LoggingQueryPostProcessor` specifically to the `CustomerRepository`, as follows:

Example 15. CustomerLoggingQueryPostProcessor

```
class LoggingQueryPostProcessor implements QueryPostProcessor<CustomerRepository, String> { .. }
```

As a result, only queries defined in the `CustomerRepository` interface, such as `findByAccountNumber`, are logged.

You might want to create a `QueryPostProcessor` for a specific query defined by a Repository query method. For example, suppose you want to limit the OQL query generated from the `CustomerRepository.findByLastNameLike(:String)` query method to only return five results along with ordering the `Customers` by `firstName`, in ascending order . To do so, you can define a custom `QueryPostProcessor`, as the following example shows:

Example 16. 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 the preceding example works, you can achieve the same effect by using the Spring Data Repository convention provided by Spring Data for Pivotal GemFire. For instance, the same query could be defined as follows:

Example 17. 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` (that is, `OrderedLimitedCustomerByLastNameQueryPostProcessor`) is convenient.

If you want to ensure that the `LoggingQueryPostProcessor` always comes after the other application-defined `QueryPostProcessors` that may have bean declared and registered in the Spring `ApplicationContext`, you can set the `order` property by overriding the `o.s.core.Ordered.getOrder()` method, as the following example shows:

Example 18. 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 you always see the effects of the post processing applied by other `QueryPostProcessors` before the `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 as granular as you like by using the provided arguments to the `postProcess(..)` method callback.

---

12. Annotation Support for Function Execution

Spring Data for Pivotal GemFire includes annotation support to simplify working with Pivotal GemFire Function execution.

Under the hood, the Pivotal GemFire API provides classes to implement and register Pivotal GemFire Functions that are deployed on Pivotal GemFire servers, which may then be invoked by other peer member applications or remotely from cache clients.

Functions can execute in parallel, distributed among multiple Pivotal GemFire servers in the cluster, aggregating the results using the map-reduce pattern and sent back to the caller. Functions can also be targeted to run on a single server or Region. The Pivotal GemFire API supports remote execution of Functions targeted by using various predefined scopes: on Region, on members (in groups), on servers, and others. The implementation and execution of remote Functions, as with any RPC protocol, requires some boilerplate code.

Spring Data for Pivotal GemFire, true to Spring's core value proposition, aims to hide the mechanics of remote Function execution and let you focus on core POJO programming and business logic. To this end, Spring Data for Pivotal GemFire introduces annotations to declaratively register the public methods of a POJO class as Pivotal GemFire Functions along with the ability to invoke registered Functions (including remotely) by using annotated interfaces.

12.1. Implementation Versus Execution

There are two separate concerns to address: implementation and execution.

The first is Function implementation (server-side), which must interact with the `FunctionContext` to access the invocation arguments, `ResultsSender` to send results, and other execution context information. The Function implementation typically accesses the cache and 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`.

'Client' and 'Server' are used here in the context of Function execution, which may have a different meaning than client and server in Pivotal GemFire's client-server topology. While it is common for an application using a `ClientCache` instance to invoke a Function on one or more Pivotal GemFire 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` instance. Keep in mind that a peer member cache application is subject to all the constraints of being a peer member of the cluster.

12.2. Implementing a Function

Using Pivotal GemFire 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, any filter (a set of specific keys) associated with the `Execution`, and so on. If the Region is a `PARTITION` Region, the Function should use the `PartitionRegionHelper` to extract the local data set.

By using Spring, you can write a simple POJO and use the Spring container to bind one or more of your POJO'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 is 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 an `int` as the calling arguments. These are provided in the `FunctionContext` as an array, as the following example shows:

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

The Spring container should be able to bind to any method signature similar to the following (ignoring 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 (that is, 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 (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 preceding example, it is also valid to pass the `FunctionContext` itself or the `ResultSender` if you need to control over how the results are returned to the client.

12.2.1. Annotations for Function Implementation

The following example shows how SDG's Function annotations are used to expose POJO methods as Pivotal GemFire Functions:

```
@Component
public class ApplicationFunctions {

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

   @GemfireFunction(id = "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 that the class itself must be registered as a Spring bean and each Pivotal GemFire Function is annotated with `@GemfireFunction`. In the preceding example, Spring's `@Component` annotation was used, but you can register the bean by using any method supported by Spring (such as XML configuration or with a Java configuration class when using Spring Boot). This lets the Spring container 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.

The fact that the POJO Function class is a Spring bean may offer other benefits. Since it shares the `ApplicationContext` with Pivotal GemFire components, such as the cache and Regions, these may be injected into the class if necessary.

Spring creates the wrapper class and registers the Functions with Pivotal GemFire's `FunctionService`. The Function ID used to register each Function must be unique. By using convention, it defaults to the simple (unqualified) method name. The name can be explicitly defined by 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 Pivotal GemFire's `Function` interface.

If the POJO Function method's return type is `void`, then the `hasResult` attribute is automatically set to `false`. Otherwise, if the method returns a value, the `hasResult` attributes is set to `true`. Even for `void` method return types, the `GemfireFunction` annotation's `hasResult` attribute can be set to `true` to override this convention, as shown in the `functionWithContext` method shown previously. Presumably, the intention is that you will use the `ResultSender` directly to send results to the caller.

Finally, the `GemfireFunction` annotation supports the `requiredPermissions` attribute, which specifies the permissions required to execute the Function. By default, all Functions require the `DATA:WRITE` permission. The attribute accepts an array of Strings allowing you to modify the permissions as required by your application and/or Function UC. Each resource permission is expected to be in the following format: `<RESOURCE>:<OPERATION>:[Target]:[Key]`.

`RESOURCE` can be 1 of the {data-store-javadoc]/org/apache/geode/security/ResourcePermission.Resource.html[`ResourcePermission.Resource`] enumerated values. `OPERATION` can be 1 of the {data-store-javadoc}/org/apache/geode/security/ResourcePermission.Operation.html[`ResourcePermission.Operation`] enumerated values. Optionally, `Target` can be the name of a Region or 1 of the {data-store-javadoc}/org/apache/geode/security/ResourcePermission.Target.html[`ResourcePermission.Target`] enumerated values. And finally, optionally, `Key` is a valid Key in the `Target` Region if specified.

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

12.2.2. 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` is quite large, it may incur a performance penalty. To divide the payload into smaller, more manageable chunks, you can set the `batchSize` attribute, as illustrated in `function2`, shown earlier.

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 in the `ResultSender` or access it through the `FunctionContext` and use it directly within the method to sends results back to the caller.

12.2.3. Enabling Annotation Processing

In accordance with Spring standards, you must explicitly activate annotation processing for `@GemfireFunction` annotations. The following example activates annotation processing with XML:

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

The following example activates annotation processing by annotating a Java configuration class:

```
@Configuration
@EnableGemfireFunctions
class ApplicationConfiguration { ... }
```

12.3. Executing a Function

A process that invokes a remote Function needs to provide the Function's ID, calling arguments, the execution target (`onRegion`, `onServers`, `onServer`, `onMember`, or `onMembers`) and (optionally) a filter set. By using Spring Data for Pivotal GemFire, all you need do is define an interface supported by annotations. Spring creates a dynamic proxy for the interface, which uses the `FunctionService` to create an `Execution`, invoke the `Execution`, and (if necessary) coerce the results to the defined return type. This technique is similar to the way Spring Data for Pivotal GemFire'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.

12.3.1. Annotations for Function Execution

To support client-side Function execution, the following SDG Function annotations are provided: `@OnRegion`, `@OnServer`, `@OnServers`, `@OnMember`, and `@OnMembers`. These annotations correspond to the `Execution` implementations provided by Pivotal GemFire's `FunctionService` class.

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` interface to use for the execution.

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

The following listing shows 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.

12.3.2. 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, insert the following element in your XML configuration:

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

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

Optionally, you can annotate your Java configuration class as follows:

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

12.4. 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. In the following example, the `GemfireOnRegionFunctionTemplate` creates an `onRegion` Function `Execution`:

Example 19. 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 attempts 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 remaining arguments are a variable argument `List`.

12.5. Function Execution with PDX

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

As explained earlier in this section, and by way of example, you should typically define Pivotal GemFire Functions by using POJO classes annotated with Spring Data for Pivotal GemFire Function annotations, as follows:

```
public class OrderFunctions {

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

}
```

The `Integer` typed `count` parameter is arbitrary, as is the separation of the `Order` class and the `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` class and `OrderSource` enum might be defined as follows:

```
public class Order ... {

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

  ...
}


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

Of course, you can define a Function `Execution` interface to call the 'process' Pivotal GemFire server Function, as follows:

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

Clearly, this `process(..)` `Order` Function is being called from the client-side with a `ClientCache` instance (that is `<gfe:client-cache/>`). This implies that the Function arguments must also be serializable. The same is true when invoking peer-to-peer member Functions (such as `@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 you have configured Pivotal GemFire to use PDX for serialization (instead of Java serialization, for instance) you can also set the `pdx-read-serialized` attribute to `true` in your configuration of the Pivotal GemFire server(s), as follows:

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

Alternatively, you can set the `pdx-read-serialized` attribute to `true` for a Pivotal GemFire cache client application, as follows:

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

Doing so causes all values read from the cache (that is, Regions) as well as information passed between client and servers (or peers) to remain in serialized form, including, but not limited to, Function arguments.

Pivotal GemFire serializes only application domain object types that you have specifically configured (registered) either by using Pivotal GemFire's `ReflectionBasedAutoSerializer`, or specifically (and recommended) by using a "custom" Pivotal GemFire `PdxSerializer`. If you use Spring Data for Pivotal GemFire's Repository extension, you might even want to consider using Spring Data for Pivotal GemFire's `MappingPdxSerializer`, which uses an entity's mapping metadata to determine data from the application domain object that is serialized to the PDX instance.

What is less than apparent, though, is that Pivotal GemFire automatically handles Java `Enum` types regardless of whether they are explicitly configured (that is, registered with a `ReflectionBasedAutoSerializer`, using a regex pattern and the `classes` parameter or are handled by a "custom" Pivotal GemFire `PdxSerializer`), despite the fact that Java enumerations implement `java.io.Serializable`.

So, when you set `pdx-read-serialized` to `true` on Pivotal GemFire servers where the Pivotal GemFire Functions (including Spring Data for Pivotal GemFire Function-annotated POJO classes) are registered, then you may encounter surprising behavior when invoking the Function `Execution`.

You might pass the following arguments when invoking the Function:

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

However, the Pivotal GemFire Function on the server gets the following:

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

The `Order` and `OrderSource` have been passed to the Function as PDX instances. Again, this all happens because `pdx-read-serialized` is set to `true`, which may be necessary in cases where the Pivotal GemFire servers interact with multiple different clients (for example, a combination of Java clients and native clients, such as C/C++, C#, and others).

This flies in the face of Spring Data for Pivotal GemFire's strongly-typed Function-annotated POJO class method signatures, where you would reasonably expect application domain object types instead, not PDX serialized instances.

Consequently, Spring Data for Pivotal GemFire includes enhanced Function support to automatically convert PDX typed method arguments to the desired application domain object types defined by the Function method's signature (parameter types).

However, this also requires you to explicitly register a Pivotal GemFire `PdxSerializer` on Pivotal GemFire servers where Spring Data for Pivotal GemFire Function-annotated POJOs are registered and used, as the following example shows:

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

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

Alternatively, you can use Pivotal GemFire's `ReflectionBasedAutoSerializer` for convenience. Of course, we recommend that, where possible, you use a custom `PdxSerializer` to maintain finer-grained control over your serialization strategy.

Finally, Spring Data for Pivotal GemFire is careful not to convert your Function arguments if you treat your Function arguments generically or as one of Pivotal GemFire's PDX types, as follows:

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

Spring Data for Pivotal GemFire converts PDX typed data to the corresponding application domain types if and only if the corresponding application domain types are on the classpath and the Function-annotated POJO method expects it.

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

---

13. Apache Lucene Integration

Pivotal GemFire integrates with Apache Lucene to let you index and search on data stored in Pivotal GemFire by using Lucene queries. Search-based queries also include the ability to page through query results.

Additionally, Spring Data for Pivotal GemFire adds support for query projections based on the Spring Data Commons projection infrastructure. This feature lets the query results be projected into first-class application domain types as needed by the application.

A Lucene `Index` must be created before any Lucene search-based query can be run. A `LuceneIndex` can be created in Spring (Data for Pivotal GemFire) XML config as follows:

```
<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 as shown in the following example:

```
<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>
```

The `Map` can be specified as a top-level bean definition and referenced by using the `ref` attribute in the nested `<gfe:field-analyzers>` element, as follows: `<gfe-field-analyzers ref="refToTopLevelMapBeanDefinition"/>`.

Spring Data for Pivotal GemFire's `LuceneIndexFactoryBean` API and SDG's XML namespace also lets a `org.apache.geode.cache.lucene.LuceneSerializer` be specified when you create the `LuceneIndex`. The `LuceneSerializer` lets you configure the way objects are converted to Lucene documents for the index when the object is indexed.

The following example shows how to add an `LuceneSerializer` to the `LuceneIndex`:

```
<bean id="MyLuceneSerializer" class="example.CustomLuceneSerializer"/>

<gfe:lucene-index id="IndexThree" lucene-service-ref="luceneService" region-path="/YetAnotherExample">
    <gfe:lucene-serializer ref="MyLuceneSerializer">
</gfe:lucene-index>
```

You can specify the `LuceneSerializer` as an anonymous, nested bean definition as well, as follows:

```
<gfe:lucene-index id="IndexThree" lucene-service-ref="luceneService" region-path="/YetAnotherExample">
    <gfe:lucene-serializer>
        <bean class="example.CustomLuceneSerializer"/>
    </gfe:lucene-serializer>
</gfe:lucene-index>
```

Alternatively, you can declare or define a `LuceneIndex` in Spring Java config, inside a `@Configuration` class, as the following example shows:

```
@Bean(name = "Books")
@DependsOn("bookTitleIndex")
PartitionedRegionFactoryBean<Long, Book> booksRegion(GemFireCache gemfireCache) {

    PartitionedRegionFactoryBean<Long, Book> peopleRegion =
        new PartitionedRegionFactoryBean<>();

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

    return peopleRegion;
}

@Bean
LuceneIndexFactoryBean bookTitleIndex(GemFireCache gemFireCache,
        LuceneSerializer luceneSerializer) {

    LuceneIndexFactoryBean luceneIndex = new LuceneIndexFactoryBean();

    luceneIndex.setCache(gemFireCache);
    luceneIndex.setFields("title");
    luceneIndex.setLuceneSerializer(luceneSerializer);
    luceneIndex.setRegionPath("/Books");

    return luceneIndex;
}

@Bean
CustomLuceneSerializer myLuceneSerialier() {
    return new CustomeLuceneSerializer();
}
```

There are a few limitations of Pivotal GemFire's, Apache Lucene integration and support.

First, a `LuceneIndex` can only be created on a Pivotal GemFire `PARTITION` Region.

Second, all `LuceneIndexes` must be created before the Region to which the `LuceneIndex` applies.

To help ensure that all declared `LuceneIndexes` defined in a Spring container are created before the Regions on which they apply, SDG includes the

`org.springframework.data.gemfire.config.support.LuceneIndexRegionBeanFactoryPostProcessor`. You may register this Spring `BeanFactoryPostProcessor` in XML config by using `<bean class="org.springframework.data.gemfire.config.support.LuceneIndexRegionBeanFactoryPostProcessor"/>`. The `o.s.d.g.config.support.LuceneIndexRegionBeanFactoryPostProcessor` may only be used when using SDG XML config. More details about Spring's `BeanFactoryPostProcessors` can be found here.

It is possible that these Pivotal GemFire 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 when you 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 adhere to strongly-typed objects. However, for the time being, you must use the `regionPath` property to specify the Region to which the `LuceneIndex` is applied.

Additionally, in the preceding example, note the presence of Spring's `@DependsOn` annotation on the `Books` Region bean definition. This creates a dependency from the `Books` Region bean to the `bookTitleIndex` `LuceneIndex` bean definition, ensuring that the `LuceneIndex` is created before the Region on which it applies.

Now that once we have a `LuceneIndex`, we can perform Lucene-based data access operations, such as queries.

13.1. Lucene Template Data Accessors

Spring Data for Pivotal GemFire provides two primary templates for Lucene data access operations, depending on how low of a level your application is prepared to deal with.

The `LuceneOperations` interface defines query operations by using Pivotal GemFire Lucene types, which are defined in the following interface definition:

```
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]);
}
```

The `[, int resultLimit]` indicates that the `resultLimit` parameter is optional.

The operations in the `LuceneOperations` interface match the operations provided by the Pivotal GemFire's LuceneQuery interface. However, SDG has the added value of translating proprietary Pivotal GemFire or Apache Lucene `Exceptions` into Spring's highly consistent and expressive DAO exception hierarchy, particularly as many modern data access operations involve more than one store or repository.

Additionally, SDG's `LuceneOperations` interface can shield your application from interface-breaking changes introduced by the underlying Pivotal GemFire or Apache Lucene APIs when they occur.

However, it would be sad to offer a Lucene Data Access Object (DAO) that only uses Pivotal GemFire and Apache Lucene data types (such as Pivotal GemFire's `LuceneResultStruct`). Therefore, SDG gives you the `ProjectingLuceneOperations` interface to remedy these important application concerns. The following listing shows the `ProjectingLuceneOperations` interface definition:

```
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 that let you 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 by using the Spring Data Commons Projection infrastructure.

Additionally, the template wraps the paged Lucene query results in an instance of the Spring Data Commons `Page` abstraction. 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 you have a class representing a `Person`, as follows:

```
class Person {

    Gender gender;

    LocalDate birthDate;

    String firstName;
    String lastName;

    ...

    String getName() {
        return String.format("%1$s %2$s", getFirstName(), getLastName());
    }
}
```

Additionally, you might have a single interface to represent people as `Customers`, depending on your application view, as follows:

```
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 you could query for people as `Person` objects, as follows:

```
List<Person> people = luceneTemplate.query("lastName: D*", "lastName", Person.class);
```

Alternatively, you could query for a `Page` of type `Customer`, as follows:

```
Page<Customer> customers = luceneTemplate.query("lastName: D*", "lastName", 100, 20, Customer.class);
```

The `Page` can then be used to fetch individual pages of the results, as follows:

```
List<Customer> firstPage = customers.getContent();
```

Conveniently, the Spring Data Commons `Page` interface also implements `java.lang.Iterable<T>`, making it easy to iterate over the contents.

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 SDC Projection infrastructure and provide a custom `ProjectionFactory` that uses CGLIB to generate proxy classes as the projected entity.

You can use `setProjectionFactory(:ProjectionFactory)` to set a custom `ProjectionFactory` on a Lucene template.

13.2. Annotation Configuration Support

Finally, Spring Data for Pivotal GemFire provides annotation configuration support for `LuceneIndexes`.

Eventually, the SDG Lucene support will finds its way into the Repository infrastructure extension for Pivotal GemFire so that Lucene queries can be expressed as methods on an application `Repository` interface, in much the same way as the OQL support works today.

However, in the meantime, if you want to conveniently express `LuceneIndexes`, you can do so directly on your application domain objects, as the following example shows:

```
@PartitionRegion("People")
class Person {

    Gender gender;

    @Index
    LocalDate birthDate;

    String firstName;

    @LuceneIndex;
    String lastName;

    ...
}
```

To enable this feature, you must use SDG's annotation configuration support specifically with the `@EnableEntityDefineRegions` and `@EnableIndexing` annotations, as follows:

```
@PeerCacheApplication
@EnableEntityDefinedRegions
@EnableIndexing
class ApplicationConfiguration {

  ...
}
```

`LuceneIndexes` can only be created on Pivotal GemFire servers since `LuceneIndexes` only apply to `PARTITION` Regions.

Given our earlier definition of the `Person` class, the SDG annotation configuration support finds the `Person` entity class definition and determines that people are stored in a `PARTITION` Region called "People" and that the `Person` has an OQL `Index` on `birthDate` along with a `LuceneIndex` on `lastName`.

---

14. Bootstrapping a Spring ApplicationContext in Pivotal GemFire

Normally, a Spring-based application bootstraps Pivotal GemFire by using Spring Data for Pivotal GemFire's features. By specifying a `<gfe:cache/>` element that uses the Spring Data for Pivotal GemFire XML namespace, a single embedded Pivotal GemFire peer `Cache` instance is created and initialized with default settings in the same JVM process as your application.

However, it is sometimes necessary (perhaps as a requirement imposed by your IT organization) that Pivotal GemFire be fully managed and operated by the provided Pivotal GemFire tool suite, perhaps using Gfsh. By using *Gfsh*, Pivotal GemFire bootstraps your Spring `ApplicationContext` rather than the other way around. Instead of an application server or a Java main class that uses Spring Boot, Pivotal GemFire does the bootstrapping and hosts your application.

Pivotal GemFire is not an application server. In addition, there are limitations to using this approach where the Pivotal GemFire cache configuration is concerned.

14.1. Using Pivotal GemFire to Bootstrap a Spring Context Started with Gfsh

In order to bootstrap a Spring `ApplicationContext` in Pivotal GemFire when starting a Pivotal GemFire server using *Gfsh*, you must use Pivotal GemFire's initalizer capability. An initializer block can declare a application callback that is launched after the cache is initialized by Pivotal GemFire.

An initializer is declared within an initializer element by using a minimal snippet of Pivotal GemFire's native `cache.xml`. To bootstrap the Spring `ApplicationContext`, a `cache.xml` file is required, in much the same way as a minimal snippet of Spring XML config is needed to bootstrap a Spring `ApplicationContext` configured with component scanning (for example `<context:component-scan base-packages="…"/>`).

Fortunately, such an initializer is already conveniently provided by the framework: the `SpringContextBootstrappingInitializer`.

The following example shows a typical, yet minimal, configuration for this class inside Pivotal GemFire's `cache.xml` 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">

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

</cache>
```

The `SpringContextBootstrappingInitializer` class follows conventions similar to Spring's `ContextLoaderListener` class, which is used to bootstrap a Spring `ApplicationContext` inside a web application, where `ApplicationContext` 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 that contain appropriately annotated application components. The Spring container searches these components to find and create Spring beans and other application components in the classpath, as the following example shows:

```
<?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</class-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` and `cache.xml` file (shown earlier) specified as a command-line option when starting a Pivotal GemFire server in *Gfsh*, the command-line would be as follows:

```
gfsh>start server --name=ExampleServer --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 configuration metadata, including all of the SDG XML namespace elements. The only limitation with this approach is that a Pivotal GemFire cache cannot be configured by using the SDG XML 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`, and others) can be specified. If used, these attributes are ignored.

The reason for this is that Pivotal GemFire itself has already created and initialized the cache before the initializer gets invoked. As a result, the cache already exists and, since it is a “singleton”, it cannot be re-initialized or have any of its configuration augmented.

14.2. Lazy-wiring Pivotal GemFire Components

Spring Data for Pivotal GemFire already provides support for auto-wiring Pivotal GemFire components (such as `CacheListeners`, `CacheLoaders`, `CacheWriters` and so on) that are declared and created by Pivotal GemFire in `cache.xml` by using SDG’s `WiringDeclarableSupport` class, as described in Configuration using auto-wiring and annotations. However, this works only when Spring is the one doing the bootstrapping (that is, when Spring bootstraps Pivotal GemFire).

When your Spring `ApplicationContext` is bootstrapped by Pivotal GemFire, these Pivotal GemFire application components go unnoticed, because the Spring `ApplicationContext` does not exist yet. The Spring `ApplicationContext` does not get created until Pivotal GemFire calls the initializer block, which only occurs after all the other Pivotal GemFire components (cache, Regions, and others) have already been created and initialized.

To solve this problem, a new `LazyWiringDeclarableSupport` class was introduced. This new class is aware of the Spring `ApplicationContext`. The intention behind this abstract base class is that any implementing class registers itself to be configured by the Spring container that is eventually created by Pivotal GemFire once the initializer is called. In essence, this gives your Pivotal GemFire application components a chance to be configured and auto-wired with Spring beans defined in the Spring container.

In order for your Pivotal GemFire application components to be auto-wired by the Spring container, you should 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 the following example:

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

  @Autowired
  private DataSource userDataSource;

  ...
}
```

As implied in the `CacheLoader` example above, you might necessarily (though rarely) have defined both a Region and a `CacheListener` component in Pivotal GemFire `cache.xml`. The `CacheLoader` may need access to an application Repository (or perhaps a JDBC `DataSource` defined in the Spring `ApplicationContext`) for loading `Users` into a Pivotal GemFire `REPLICATE` Region on startup.

CAUTION

Be careful when mixing the different life-cycles of Pivotal GemFire and the Spring container together in this manner. Not all use cases and scenarios are supported. The Pivotal GemFire `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.SpringContextBootstrappingInitializerIntegrationTest$UserDataStoreCacheLoader
        </class-name>
      </cache-loader>
    </region-attributes>
  </region>

  <initializer>
    <class-name>org.springframework.data.gemfire.support.SpringContextBootstrappingInitializer</class-name>
    <parameter name="basePackages">
      <string>org.springframework.data.gemfire.support.sample</string>
    </parameter>
  </initializer>

</cache>
```

15. Sample Applications

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

The Spring Data for Pivotal GemFire project also includes one sample application. Named "Hello World", the sample application demonstrates how to configure and use Pivotal GemFire inside a Spring application. At run time, the sample offers a shell that lets you run various commands against the data grid. It provides an excellent starting point for developers who are unfamiliar with the essential components or with Spring and Pivotal GemFire concepts.

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

15.1. Hello World

The "Hello World" sample application demonstrates the core functionality of the Spring Data for Pivotal GemFire project. It bootstraps Pivotal GemFire, 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 work together, sharing data without any user intervention.

Running under Linux

If you experience networking problems when starting Pivotal GemFire or the samples, try adding the following system property `java.net.preferIPv4Stack=true` to the command line (for example, `-Djava.net.preferIPv4Stack=true`). For an alternative (global) fix (especially on Ubuntu), see SGF-28.

15.1.1. Starting and Stopping the Sample

The "Hello World" sample application is designed as a stand-alone Java application. It features a `main` class that can be started either from your IDE (in Eclipse or STS, through `Run As/Java Application`) or from the command line through Maven with `mvn exec:java`. If the classpath is properly set, you can also use `java` directly on the resulting artifact.

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

15.1.2. Using the Sample

Once started, the sample creates a shared data grid and lets you issue commands against it. The output should resemble the following:

```
INFO: Created {data-store-name} Cache [Spring {data-store-name} 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, you can use the following commands:

```
-> 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, as the following example shows:

```
INFO: Connected to Distributed System ['Spring {data-store-name} 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]
```

We encourage you to experiment with the example, start (and stop) as many instances as you want, and 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.

15.1.3. Hello World Sample Explained

The “Hello World” sample 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 Pivotal GemFire cache, a region, and for illustrative purposes, a `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 {data-store-product-name} with the *Spring Framework*. These additional, third-party resources are enumerated in this section.

16. Useful Links

- Spring Data for Pivotal GemFire Project Page
- Spring Data for Pivotal GemFire source code
- Spring Data for Pivotal GemFire JIRA
- Spring Data for Pivotal GemFire on StackOverflow
- Archive of the Spring Data for Pivotal GemFire Forum on Spring IO
- Pivotal GemFire Home Page
- Pivotal GemFire Documentation
- Apache Geode Community
- Apache Geode source code
- Apache Geode JIRA
- Pivotal GemFire 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. See "[repositories.create-instances.spring]". The following table describes the attributes of the `<repositories />` element:

Table 6. Attributes

Name	Description
`base-package`	Defines the package to be scanned for repository interfaces that extend `*Repository` (the actual interface is determined by the specific Spring Data module) in auto-detection mode. All packages below the configured package are scanned, too. Wildcards are allowed.
`repository-impl-postfix`	Defines the postfix to autodetect custom repository implementations. Classes whose names end with the configured postfix are 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 search for a Properties file containing externally defined queries.
`consider-nested-repositories`	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.[1]

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 keywords listed here might not be supported in a particular store.

Table 8. Query keywords

Logical keyword	Keyword expressions
`AND`	`And`
`OR`	`Or`

Logical keyword	Keyword expressions
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 types listed here might not be supported in a particular store.

Geospatial types (such as `GeoResult`, `GeoResults`, and `GeoPage`) are available only 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. If no result is found, `null` is returned. More than one result triggers 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. If no result is found, `Optional.empty()` or `Optional.absent()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Option<T>`	Either a Scala or Javaslang `Option` type. Semantically the same behavior as Java 8's `Optional`, described earlier.
`Stream<T>`	A Java 8 `Stream`.
`Future<T>`	A `Future`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`CompletableFuture<T>`	A Java 8 `CompletableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.
`ListenableFuture`	A `org.springframework.util.concurrent.ListenableFuture`. Expects a method to be annotated with `@Async` and requires Spring's asynchronous method execution capability to be enabled.

Return type	Description
`Slice`	A sized chunk of data with an indication of whether there is more data available. Requires a `Pageable` method parameter.
`Page<T>`	A `Slice` with additional information, such as the total number of results. Requires a `Pageable` method parameter.
`GeoResult<T>`	A result entry with additional information, such as the distance to a reference location.
`GeoResults<T>`	A list of `GeoResult<T>` with additional information, such as the average distance to a reference location.
`GeoPage<T>`	A `Page` with `GeoResult<T>`, such as the average distance to a reference location.
`Mono<T>`	A Project Reactor `Mono` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flux<T>`	A Project Reactor `Flux` emitting zero, one, or many elements using reactive repositories. Queries returning `Flux` can emit also an infinite number of elements.
`Single<T>`	A RxJava `Single` emitting a single element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Maybe<T>`	A RxJava `Maybe` emitting zero or one element using reactive repositories. Expects the query method to return one result at most. If no result is found, `Mono.empty()` is returned. More than one result triggers an `IncorrectResultSizeDataAccessException`.
`Flowable<T>`	A RxJava `Flowable` emitting zero, one, or many elements using reactive repositories. Queries returning `Flowable` can emit also an infinite number of elements.

Appendix E: Spring Data for Pivotal GemFire Schema

- Spring Data for Pivotal GemFire Core Schema (`gfe` XML namespace)
- Spring Data for Pivotal GemFire Data Access Schema (`gfe-data` XML namespace)

1. see [repositories.create-instances.spring]

Version 2.1.1.RELEASE
Last updated 2018-10-15 10:39:25 MESZ