

Spring Boot for Apache Geode Reference Guide

John Blum

Version 1.5.0-RC1, 2021-04-28

Table of Contents

1. Introduction	2
2. Getting Started	4
3. Using Spring Boot for Apache Geode	5
4. Building ClientCache Applications	10
5. Auto-configuration	29
6. Declarative Configuration	42
7. Externalized Configuration	57
8. Using Geode Properties	60
9. Caching with Apache Geode	67
10. Data Access with GemfireTemplate	89
11. Spring Data Repositories	95
12. Function Implementations & Executions	97
13. Continuous Query	100
14. Using Data	102
15. Data Serialization with PDX	117
16. Logging	120
17. Security	129
18. Testing	134
19. Apache Geode API Extensions	140
20. Spring Boot Actuator	149
21. Spring Session	167
22. Pivotal CloudFoundry	173
23. Docker	194
24. Samples	203
25. Appendix	205

Welcome to *Spring Boot for Apache Geode & VMware Tanzu GemFire*.

Spring Boot for Apache Geode provides the convenience of Spring Boot's *convention over configuration* approach using *auto-configuration* with the Spring Framework's powerful abstractions and highly consistent programming model to truly simplify the development of Apache Geode or VMware Tanzu GemFire applications in a Spring context.

Secondarily, Spring Boot for Apache Geode & VMware Tanzu GemFire aims to provide developers with a consistent experience whether building and running Spring Boot, Apache Geode & VMware Tanzu GemFire applications locally or in a managed environment, such as with [Pivotal CloudFoundry](#) (PCF).

This project is a continuation and a logical extension to Spring Data for Apache Geode & VMware Tanzu GemFire's [Annotation-based configuration model](#) and the goals set forth in that model: *To enable application developers to **get up and running** as **quickly** and as **easily** as possible*. In fact, Spring Boot for Apache Geode & VMware Tanzu GemFire builds on this very [foundation](#) cemented in Spring Data for Apache Geode & VMware Tanzu GemFire (SDG ^[1]) since the Spring Data Kay Release Train.

Chapter 1. Introduction

Spring Boot for Apache Geode & VMware Tanzu GemFire automatically applies *auto-configuration* to several key application concerns (*Use Cases*) including, but not limited to:

- *Look-Aside, [Async] Inline, Near and Multi-Site Caching*, using Apache Geode as a caching provider in [Spring's Cache Abstraction](#). [Learn more](#).
- *System of Record (SOR)*, persisting application state reliably in Apache Geode using [Spring Data Repositories](#). [Learn more](#).
- *Transactions*, managing application state consistently with [Spring Transaction Management](#) and SDG^[1] with support for both [Local Cache](#) and [Global JTA](#) Transactions.
- *Distributed Computations*, ran with Apache Geode's [Function Execution](#) framework, and conveniently implemented and executed with SDG^[1] [POJO-based, annotation support for Functions](#). [Learn more](#).
- *Continuous Queries*, expressing interests in a stream of events, allowing applications to react to and process changes to data in near real-time with Apache Geode's [Continuous Query \(CQ\)](#). Handlers are defined as simple Message-Driven POJOs (MDP) using Spring's [Message Listener Container](#), which has been [extended](#) in SDG^[1] with its [configurable CQ](#) support. [Learn more](#).
- *Data Serialization* with Apache Geode [PDX](#), including first-class [configuration](#) and [support](#) in SDG^[1]. [Learn more](#).
- *Data Initialization* to quickly load (import) to hydrate the cache during application startup or write (export) data on application shutdown to move data between environments (e.g. TEST to DEV). [Learn more](#).
- *Actuator* - to gain insight into the runtime behavior and operation of your cache, whether a client or a peer. [Learn more](#).
- *Logging* - quickly and conveniently enable or adjust Apache Geode log levels in your Spring Boot application to gain insight into the runtime operations of the app as they occur. [Learn more](#).
- *Security*, including [Authentication](#) & [Authorization](#) as well as Transport Layer Security (TLS) using Apache Geode [Secure Socket Layer \(SSL\)](#). Once more, SDG^[1] includes first-class support for configuring [Auth](#) and [SSL](#). [Learn more](#).
- *HTTP Session state management*, by including Spring Session for Apache Geode on your application's classpath. [Learn more](#).
- *Testing* - whether writing Unit or Integration Tests for Apache Geode in a Spring context, SBDG covers all your testing needs with the help of [STDG](#).

While Spring Data for Apache Geode & VMware Tanzu GemFire offers a simple, consistent, convenient and declarative approach to configure all these powerful Apache Geode features, Spring Boot for Apache Geode & VMware Tanzu GemFire makes it even easier to do as we will explore throughout this reference documentation.

1.1. Goals

While the SBDG project has many goals and objectives, the primary goals of this project centers around three key principles:

1. From ***Open Source*** (Apache Geode) to ***Commercial*** (VMware Tanzu GemFire)
2. From ***Non-Managed*** (self-managed/hosted or on-premise installations) to ***Managed*** (VMware Tanzu GemFire for VMs, VMware Tanzu GemFire for K8S) environments
3. With **little to no code or configuration changes** necessary

It is also possible to go in the reverse direction, from *Managed* back to a *Non-Managed* environment and even from *Commercial* back to the *Open Source* offering, again, with *little to no code or configuration* changes; it simply just works!



SBDG's promise is to deliver on these principles as much as is technically possible and is technically allowed by Apache Geode.

[1] Spring Data for Apache Geode and Spring Data for VMware Tanzu GemFire are commonly known as SDG.

Chapter 2. Getting Started

In order to be immediately productive and as effective as possible using Spring Boot for Apache Geode & VMware Tanzu GemFire, it is helpful to understand the foundation on which this project was built.

Of course, our story begins with the Spring Framework and the [core technologies and concepts](#) built into the Spring container.

Then, our journey continues with the extensions built into Spring Data for Apache Geode & VMware Tanzu GemFire (SDG^[1]) to truly simplify the development of Apache Geode & VMware Tanzu GemFire applications in a Spring context, using Spring's powerful abstractions and highly consistent programming model. This part of the story was greatly enhanced in Spring Data Kay, with the SDG^[1] [Annotation-based configuration model](#). Though this new configuration approach using annotations provides sensible defaults out-of-the-box, its use is also very explicit and assumes nothing. If any part of the configuration is ambiguous, SDG will fail fast. SDG gives you "*choice*", so you still must tell SDG^[1] what you want.

Next, we venture into Spring Boot and all of its wonderfully expressive and highly opinionated "*convention over configuration*" approach for getting the most out of your Spring, Apache Geode/VMware Tanzu GemFire based applications in the easiest, quickest and most reliable way possible. We accomplish this by combining Spring Data for Apache Geode & VMware Tanzu GemFire's [Annotation-based configuration](#) with Spring Boot's [auto-configuration](#) to get you up and running even faster and more reliably so that you are productive from the start.

As such, it would be pertinent to begin your Spring Boot education [here](#).

Finally, we arrive at Spring Boot for Apache Geode & VMware Tanzu GemFire (SBDG).



Refer to the corresponding Sample [Guide](#) and [Code](#) to see Spring Boot for Apache Geode in action!

Chapter 3. Using Spring Boot for Apache Geode

To use Spring Boot for Apache Geode, simply declare the `spring-geode-starter` on your Spring Boot application classpath:

Maven

```
<dependencies>
  <dependency>
    <groupId>org.springframework.geode</groupId>
    <artifactId>spring-geode-starter</artifactId>
    <version>1.5.0-RC1</version>
  </dependency>
</dependencies>
```

Gradle

```
dependencies {
    compile 'org.springframework.geode:spring-geode-starter:1.5.0-RC1'
}
```

3.1. Maven BOM

If you anticipate using more than 1 Spring Boot for Apache Geode (SBDG) module in your Spring Boot application, then you can also use the new `org.springframework.geode:spring-geode-bom` Maven BOM in your application Maven POM.

Your application use case(s) may require more than 1 module if, for example, you need (HTTP) Session state management and replication (e.g. `spring-geode-starter-session`), or you need to enable Spring Boot Actuator endpoints for Apache Geode (e.g. `spring-geode-starter-actuator`), or perhaps you need assistance writing complex Unit and (distributed) Integration Tests using Spring Test for Apache Geode (STDG) (e.g. `spring-geode-starter-test`).

You can declare (include) and use any 1 of the SBDG modules:

- `spring-geode-starter`
- `spring-geode-starter-actuator`
- `spring-geode-starter-logging`
- `spring-geode-starter-session`
- `spring-geode-starter-test`

When more than 1 SBDG module is in play, then it makes sense to use the `spring-geode-bom` to manage all the dependencies so that the versions and transitive dependencies necessarily align properly.

Your Spring Boot application Maven POM using the `spring-geode-bom` along with 2 or more module dependencies might appear as follows:

Spring Boot application Maven POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0">

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.0-RC1</version>
  </parent>

  <artifactId>my-spring-boot-application</artifactId>

  <properties>
    <spring-geode.version>1.5.0-RC1</spring-geode.version>
  </properties>

  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.springframework.geode</groupId>
        <artifactId>spring-geode-bom</artifactId>
        <version>${spring-geode.version}</version>
        <scope>import</scope>
        <type>pom</type>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.springframework.geode</groupId>
      <artifactId>spring-geode-starter</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.geode</groupId>
      <artifactId>spring-geode-starter-session</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.geode</groupId>
      <artifactId>spring-geode-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```


Notice that 1) the Spring Boot application Maven POM (`pom.xml`) contains a `<dependencyManagement>` section declaring the `org.springframework.geode:spring-geode-bom` and that 2) none of the `spring-geode-starter[-xyz]` dependencies explicitly specify a `<version>`, which is 3) managed by the `spring-geode.version` property making it easy to switch between versions of SBDG as needed, applied evenly to all the SBDG modules declared and used in your application Maven POM.

If you change the version of SBDG, make sure to change the `org.springframework.boot:spring-boot-starter-parent` POM version to match. SBDG is always 1 `major` version behind, but matches on `minor` version and `patch` version (and `version qualifier`, e.g. `SNAPSHOT`, `M#`, `RC#`, or `RELEASE`, if applicable).

For example, SBDG `1.4.0` is based on Spring Boot `2.4.0`. SBDG `1.3.5.RELEASE` is based on Spring Boot `2.3.5.RELEASE` and so on. It is important that the versions align.

Of course, all of these concerns are easy to do and handled for you by simply going to start.spring.io and adding the "Spring for Apache Geode" dependency.

Clink on this [link](#) to get started!

3.2. Gradle Dependency Management

The user experience when using Gradle is similar to that of Maven.

Again, if you will be declaring and using more than 1 SBDG module in your Spring Boot application, for example, the `spring-geode-starter` along with the `spring-geode-starter-actuator` dependency, then using the `spring-geode-bom` inside your application Gradle build file will help.

Your application Gradle build file configuration will roughly appear as follows:

```
plugins {
    id 'org.springframework.boot' version '2.5.0-RC1'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'java'
}

// ...

ext {
    set('springGeodeVersion', "1.5.0-RC1")
}

dependencies {
    implementation 'org.springframework.geode:spring-geode-starter'
    implementation 'org.springframework.geode:spring-geode-starter-actuator'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

dependencyManagement {
    imports {
        mavenBom "org.springframework.geode:spring-geode-bom:${springGeodeVersion}"
    }
}
```

A combination of the [Spring Boot Gradle Plugin](#) and the [Spring Dependency Management Gradle Plugin](#) manage the application dependencies for you.

In a nutshell, the *Spring Dependency Management Gradle Plugin* provides dependency management capabilities for Gradle much like Maven. The *Spring Boot Gradle Plugin* defines a curated and tested set of versions for many 3rd party Java libraries. Together they make adding dependencies and managing (compatible) versions easier!

Again, you don't need to explicitly declare the version when adding a dependency, including a new SBDG module dependency (e.g. `spring-geode-starter-session`) since this has already been determined for you. You can simply declare the dependency:

```
implementation 'org.springframework.geode:spring-geode-starter-session'
```

The version of SBDG is controlled by the extension property (`springGeodeVersion`) in the application Gradle build file.

To use a different version of SBDG, simply set the `springGeodeVersion` property to the desired version (e.g. `1.3.5.RELEASE`). Of course, make sure the version of Spring Boot matches!

SBDG is always 1 **major** version behind, but matches on **minor** version and **patch** version (and **version qualifier**, e.g. `SNAPSHOT`, `M#`, `RC#`, or `RELEASE`, if applicable). For example, SBDG `1.4.0` is based on Spring Boot `2.4.0`. SBDG `1.3.5.RELEASE` is based on Spring Boot `2.3.5.RELEASE` and so on. It is

important that the versions align.

Of course, all of these concerns are easy to do and handled for you by simply going to start.spring.io and adding the "Spring for Apache Geode" dependency.

Click on this [link](#) to get started!

3.3. Repository declaration

Since you are using a Milestone version, you need to add the Spring Milestone Maven Repository.

If you are using *Maven*, include the following **repository** declaration in your **pom.xml**:

Maven

```
<repositories>
  <repository>
    <id>spring-milestone</id>
    <url>https://repo.spring.io/milestone</url>
  </repository>
</repositories>
```

If you are using *Gradle*, include the following **repository** declaration in your **build.gradle**:

Gradle

```
repositories {
  maven { url: 'https://repo.spring.io/milestone' }
}
```

Chapter 4. Building **ClientCache** Applications

The first opinionated option provided to you by Spring Boot for Apache Geode (SBDG) out-of-the-box is a **ClientCache** instance simply by declaring Spring Boot for Apache Geode on your application classpath.

It is assumed that most application developers using Spring Boot to build applications backed by Apache Geode will be building cache client applications deployed in an Apache Geode **Client/Server Topology**. The *client/server topology* is the most common and traditional architecture employed by enterprise applications when using Apache Geode.

For example, you can begin building a Spring Boot, Apache Geode **ClientCache** application by declaring the **spring-geode-starter** on your application's classpath:

Spring Boot for Apache Geode on the application classpath

```
<dependency>
  <groupId>org.springframework.geode</groupId>
  <artifactId>spring-geode-starter</artifactId>
</dependency>
```

Then, you configure and bootstrap your Spring Boot, Apache Geode **ClientCache** application with the following main application class:

*Spring Boot, Apache Geode **ClientCache** Application*

```
@SpringBootApplication
public class SpringBootApacheGeodeClientCacheApplication {

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

Your application now has a **ClientCache** instance, which is able to connect to an Apache Geode server running on **localhost**, listening on the default **CacheServer** port, **40404**.

By default, an Apache Geode server (i.e. **CacheServer**) must be running in order to use the **ClientCache** instance. However, it is perfectly valid to create a **ClientCache** instance and perform data access operations using **LOCAL** Regions. This is very useful during development.



To develop with **LOCAL** Regions, you only need to configure your cache Regions with the **ClientRegionShortcut.LOCAL** data management policy.

When you are ready to switch from your local development environment (IDE) to a client/server architecture in a managed environment, you simply change the data management policy of the client Region from **LOCAL** back to the default **PROXY**, or even a **CACHING_PROXY**, which will cause the

data to be sent/received to and from 1 or more servers, respectively.



Compare and contrast the above configuration with Spring Data for Apache Geode [approach](#).

It is uncommon to ever need a direct reference to the `ClientCache` instance provided by SBDG injected into your application components (e.g. `@Service` or `@Repository` beans defined in a Spring `ApplicationContext`) whether you are configuring additional Apache Geode objects (e.g. Regions, Indexes, etc) or simply using those objects indirectly in your applications. However, it is also possible to do so if and when needed.

For example, perhaps you want to perform some additional `ClientCache` initialization in a Spring Boot `ApplicationRunner` on startup:

Injecting a `GemFireCache` reference

```
@SpringBootApplication
public class SpringBootApacheGeodeClientCacheApplication {

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

    @Bean
    ApplicationRunner runAdditionalClientCacheInitialization(GemFireCache
gemfireCache) {

        return args -> {

            ClientCache clientCache = (ClientCache) gemfireCache;

            // perform additional ClientCache initialization as needed
        };
    }
}
```

4.1. Building Embedded (Peer & Server) Cache Applications

What if you want to build an embedded, peer `Cache` application instead?

Perhaps you need an actual peer cache member, configured and bootstrapped with Spring Boot, along with the ability to join this member to an existing cluster (of data servers) as a peer node. Well, you can do that too.

Remember the 2nd goal in Spring Boot's [documentation](#):

Be opinionated out of the box but get out of the way quickly as requirements start to diverge from the defaults.

It is the 2nd part, "*get out of the way quickly as requirements start to diverge from the defaults*" that we refer to here.

If your application requirements demand you use Spring Boot to configure and bootstrap an embedded, peer `Cache` instance, then simply declare your intention with either SDG's `@PeerCacheApplication` annotation, or alternatively, if you need to enable connections from `ClientCache` apps as well, use SDG's `@CacheServerApplication` annotation:

Spring Boot, Apache Geode `CacheServer` Application

```
@SpringBootApplication
@CacheServerApplication(name = "SpringBootApacheGeodeCacheServerApplication")
public class SpringBootApacheGeodeCacheServerApplication {

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



An Apache Geode "server" is not necessarily a `CacheServer` capable of serving cache clients. It is merely a peer member node in an Apache Geode cluster (a.k.a. distributed system) that stores and manages data.

By explicitly declaring the `@CacheServerApplication` annotation, you are telling Spring Boot that you do not want the default, `ClientCache` instance, but rather an embedded, peer `Cache` instance with a `CacheServer` component, which enables connections from `ClientCache` apps.

You can also enable 2 other Apache Geode services, an embedded *Locator*, which allows clients or even other peers to "locate" servers in the cluster, as well as an embedded *Manager*, which allows the Apache Geode application process to be managed and monitored using [Gfsh](#), Apache Geode's command-line shell tool:

```
@SpringBootApplication
@CacheServerApplication(name = "SpringBootApacheGeodeCacheServerApplication")
@EnableLocator
@EnableManager
public class SpringBootApacheGeodeCacheServerApplication {

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

Then, you can use *Gfsh* to connect to and manage this server:

```
$ echo $GEMFIRE
/Users/jblum/pivdev/apache-geode-1.2.1

$ gfsh

-----
 /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  /
 /  /  /  /  /  /  /  /  /  / 1.2.1
```

Monitor and Manage Apache Geode

```
gfsh>connect
Connecting to Locator at [host=localhost, port=10334] ..
Connecting to Manager at [host=10.0.0.121, port=1099] ..
Successfully connected to: [host=10.0.0.121, port=1099]

gfsh>list members

Name | Id
-----|-----
SpringBootApacheGeodeCacheServerApplication |
10.0.0.121(SpringBootApacheGeodeCacheServerApplication:29798)<ec><v0>:1024

gfsh>
gfsh>describe member --name=SpringBootApacheGeodeCacheServerApplication
Name      : SpringBootApacheGeodeCacheServerApplication
Id        :
10.0.0.121(SpringBootApacheGeodeCacheServerApplication:29798)<ec><v0>:1024
Host      : 10.0.0.121
Regions   :
PID       : 29798
Groups    :
Used Heap : 168M
Max Heap  : 3641M
Working Dir : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Log file   : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Locators   : localhost[10334]

Cache Server Information
Server Bind      :
Server Port     : 40404
Running         : true
Client Connections : 0
```

You can even start additional servers in *Gfsh*, which will connect to your Spring Boot configured and bootstrapped Apache Geode **CacheServer** application. These additional servers started in *Gfsh* know about the Spring Boot, Apache Geode server because of the embedded *Locator* service, which

is running on **localhost**, listening on the default *Locator* port, **10334**:

```
gfsh>start server --name=GfshServer --log-level=config --disable-default-server
Starting a Geode Server in /Users/jblum/pivdev/lab/GfshServer...
...
Server in /Users/jblum/pivdev/lab/GfshServer on 10.0.0.121 as GfshServer is currently
online.
Process ID: 30031
Uptime: 3 seconds
Geode Version: 1.2.1
Java Version: 1.8.0_152
Log File: /Users/jblum/pivdev/lab/GfshServer/GfshServer.log
JVM Arguments: -Dgemfire.default.locators=10.0.0.121:127.0.0.1[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/jblum/pivdev/apache-geode-1.2.1/lib/geode-core-
1.2.1.jar:/Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-dependencies.jar

gfsh>list members

      Name                                     | Id
-----|-----
SpringBootApacheGeodeCacheServerApplication |
10.0.0.121(SpringBootApacheGeodeCacheServerApplication:29798)<ec><v0>:1024
GfshServer                                   | 10.0.0.121(GfshServer:30031)<v1>:1025
```

Perhaps you want to start the other way around. As developer, I may need to connect my Spring Boot configured and bootstrapped Apache Geode server application to an existing cluster. You can start the cluster in *Gfsh* by executing the following commands:

```

gfsh>start locator --name=GfshLocator --port=11235 --log-level=config
Starting a Geode Locator in /Users/jblum/pivdev/lab/GfshLocator...
...
Locator in /Users/jblum/pivdev/lab/GfshLocator on 10.0.0.121[11235] as GfshLocator is
currently online.
Process ID: 30245
Uptime: 3 seconds
Geode Version: 1.2.1
Java Version: 1.8.0_152
Log File: /Users/jblum/pivdev/lab/GfshLocator/GfshLocator.log
JVM Arguments: -Dgemfire.log-level=config -Dgemfire.enable-cluster-configuration=true
-Dgemfire.load-cluster-configuration-from-dir=false
-Dgemfire.launcher.registerSignalHandlers=true -Djava.awt.headless=true
-Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-core-
1.2.1.jar:/Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-dependencies.jar

Successfully connected to: JMX Manager [host=10.0.0.121, port=1099]

Cluster configuration service is up and running.

gfsh>start server --name=GfshServer --log-level=config --disable-default-server
Starting a Geode Server in /Users/jblum/pivdev/lab/GfshServer...
....
Server in /Users/jblum/pivdev/lab/GfshServer on 10.0.0.121 as GfshServer is currently
online.
Process ID: 30270
Uptime: 4 seconds
Geode Version: 1.2.1
Java Version: 1.8.0_152
Log File: /Users/jblum/pivdev/lab/GfshServer/GfshServer.log
JVM Arguments: -Dgemfire.default.locators=10.0.0.121[11235] -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/jblum/pivdev/apache-geode-1.2.1/lib/geode-core-
1.2.1.jar:/Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-dependencies.jar

gfsh>list members
  Name      | Id
-----|-----
GfshLocator | 10.0.0.121(GfshLocator:30245:locator)<ec><v0>:1024
GfshServer  | 10.0.0.121(GfshServer:30270)<v1>:1025

```

Then, modify the `SpringBootApacheGeodeCacheServerApplication` class to connect to the existing cluster, like so:

```
@SpringBootApplication
@CacheServerApplication(name = "SpringBootApacheGeodeCacheServerApplication", locators
= "localhost[11235]")
public class SpringBootApacheGeodeCacheServerApplication {

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



Notice I configured the **SpringBootApacheGeodeCacheServerApplication** class, **@CacheServerApplication** annotation's **locators** property with the host and port (i.e. "localhost[11235]") on which I started my *Locator* using *Gfsh*.

After running your Spring Boot, Apache Geode **CacheServer** application again, and then running **list members** in *Gfsh*, you should see:

```

gfsh>list members

```

Name	Id
GfshLocator	10.0.0.121(GfshLocator:30245:locator)<ec><v0>:1024
GfshServer	10.0.0.121(GfshServer:30270)<v1>:1025
SpringBootApacheGeodeCacheServerApplication	10.0.0.121(SpringBootApacheGeodeCacheServerApplication:30279)<v2>:1026

```

gfsh>describe member --name=SpringBootApacheGeodeCacheServerApplication
Name      : SpringBootApacheGeodeCacheServerApplication
Id        : 10.0.0.121(SpringBootApacheGeodeCacheServerApplication:30279)<v2>:1026
Host      : 10.0.0.121
Regions   :
PID       : 30279
Groups    :
Used Heap : 165M
Max Heap  : 3641M
Working Dir : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Log file   : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Locators   : localhost[11235]

Cache Server Information
Server Bind      :
Server Port     : 40404
Running         : true
Client Connections : 0

```

In both scenarios, the Spring Boot configured and bootstrapped Apache Geode server and the *Gfsh Locator* and *Gfsh Server* formed a cluster.

While you can use either approach and Spring does not care, it is far more convenient to use Spring Boot and your IDE to form a small cluster while developing. By leveraging Spring profiles, it is far simpler and much faster to configure and start a small cluster.

Plus, this is useful for rapidly prototyping, testing and debugging your entire, end-to-end application and system architecture, all right from the comfort and familiarity of your IDE of choice. No additional tooling (e.g. *Gfsh*) or knowledge is required to get started quickly and easily.

Just *build* and *run*!



Be careful to vary your port numbers for the embedded services, like the *CacheServer*, *Locators* and *Manager*, especially if you start multiple instances, otherwise you will run into a `java.net.BindException` due to port conflicts.



See the Appendix, [Running an Apache Geode cluster using Spring Boot from your IDE](#) for more details.

4.2. Building Locator Applications

In addition to `ClientCache`, `CacheServer` and peer `Cache` applications, SDG, and by extension SBDG, now supports Locator-based, Spring Boot applications.

An Apache Geode Locator is a location-based service, or alternatively and more typically, a standalone process enabling clients to "locate" a cluster of Apache Geode servers to manage data. Many cache clients can connect to the same cluster in order to share data. Running multiple clients is common in a Microservices architecture where you need to scale-up the number of app instances to satisfy the demand.

A Locator is also used by joining members of an existing cluster to scale-out and increase capacity of the logically pooled system resources (i.e. Memory, CPU and Disk). A Locator maintains metadata that is sent to the clients to enable capabilities like single-hop data access, routing data access operations to the data node in the cluster maintaining the data of interests. A Locator also maintains load information for servers in the cluster, which enables the load to be uniformly distributed across the cluster while also providing fail-over services to a redundant member if the primary fails. A Locator provides many more benefits and you are encouraged to read the [documentation](#) for more details.

As shown above, a Locator service can be embedded within either a peer `Cache` or `CacheServer`, Spring Boot application using the SDG `@EnableLocator` annotation:

Embedded Locator Service

```
@SpringBootApplication
@CacheServerApplication
@EnableLocator
class SpringBootCacheServerWithEmbeddedLocatorApplication {
    // ...
}
```

However, it is more common to start standalone Locator JVM processes. This is useful when you want to increase the resiliency of your cluster in face of network and process failures, which are bound to happen. If a Locator JVM process crashes or gets severed from the cluster due to a network failure, then having multiple Locators provides a higher degree of availability (HA) through redundancy.

Not to worry though, if all Locators in the cluster go down, then the cluster will still remain intact. You simply won't be able to add more peer members (i.e. scale-up the number of data nodes in the cluster) or connect any more clients. If all the Locators in the cluster go down, then it is safe to simply restart them only after a thorough diagnosis.



Once a client receives metadata about the cluster of servers, then all data access operations are sent directly to servers in the cluster, not a Locator. Therefore, existing, connected clients will remain connected and operable.

To configure and bootstrap Locator-based, Spring Boot applications as standalone JVM processes, use the following configuration:

Standalone Locator Process

```
@SpringBootApplication
@LocatorApplication
class SpringBootApacheGeodeLocatorApplication {
    // ...
}
```

Instead of using the `@EnableLocator` annotation, you now use the `@LocatorApplication` annotation.

The `@LocatorApplication` annotation works in the same way as the `@PeerCacheApplication` and `@CacheServerApplication` annotations, bootstrapping an Apache Geode process, overriding the default `ClientCache` instance provided by SBDG out-of-the-box.



If your `@SpringBootApplication` class is annotated with `@LocatorApplication`, then it can only be a `Locator` and not a `ClientCache`, `CacheServer` or peer `Cache` application. If you need the application to function as a peer `Cache`, perhaps with an embedded `CacheServer` components and embedded Locator, then you need to follow the approach shown above using the `@EnableLocator` annotation with either the `@PeerCacheApplication` or `@CacheServerApplication` annotation.

With our Spring Boot, Apache Geode Locator application, we can connect both Spring Boot configured and bootstrapped peer members (peer `Cache`, `CacheServer` and `Locator` applications) as well as *Gfsh* started Locators and Servers.

First, let's startup 2 Locators using our Apache Geode Locator, Spring Boot application class.

```
@UseLocators
@SpringBootApplication
@LocatorApplication(name = "SpringBootApacheGeodeLocatorApplication")
public class SpringBootApacheGeodeLocatorApplication {

    public static void main(String[] args) {

        new SpringApplicationBuilder(SpringBootApacheGeodeLocatorApplication.class)
            .web(WebApplicationType.NONE)
            .build()
            .run(args);

        System.err.println("Press <enter> to exit!");

        new Scanner(System.in).nextLine();
    }

    @Configuration
    @EnableManager(start = true)
    @Profile("manager")
    @SuppressWarnings("unused")
    static class ManagerConfiguration { }

}
```

We also need to vary the configuration for each Locator app instance.

Apache Geode requires each peer member in the cluster to be uniquely named. We can set the name of the Locator by using the `spring.data.gemfire.locator.name` SDG property set as a JVM System Property in your IDE's Run Configuration Profile for the application main class like so: `-Dspring.data.gemfire.locator.name=SpringLocatorOne`. We name the second Locator app instance, `"SpringLocatorTwo"`.

Additionally, we must vary the port numbers that the Locators use to listen for connections. By default, an Apache Geode Locator listens on port `10334`. We can set the Locator port using the `spring.data.gemfire.locator.port` SDG property.

For our first Locator app instance (i.e. `"SpringLocatorOne"`), we also enable the `"manager"` Profile so that we can connect to the Locator using `Gfsh`.

Our IDE Run Configuration Profile for our first Locator app instance appears as:

```
-server -ea -Dspring.profiles.active=manager
-Dspring.data.gemfire.locator.name=SpringLocatorOne -Dlogback.log.level=INFO
```

And our IDE Run Configuration Profile for our second Locator app instance appears as:

```
-server -ea -Dspring.profiles.active= -Dspring.data.gemfire.locator.name=SpringLocatorTwo
-Dspring.data.gemfire.locator.port=11235 -Dlogback.log.level=INFO
```

You should see log output similar to the following when you start a Locator app instance:

Spring Boot, Apache Geode Locator log output

[illegible]

```

2019-09-01 11:02:48,707 INFO .SpringBootApacheGeodeLocatorApplication: 55 - Starting
SpringBootApacheGeodeLocatorApplication on jblum-mbpro-2.local with PID 30077
(/Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/out/production/classes
started by jblum in /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
docs/build)
2019-09-01 11:02:48,711 INFO .SpringBootApacheGeodeLocatorApplication: 651 - No
active profile set, falling back to default profiles: default
2019-09-01 11:02:49,374 INFO xt.annotation.ConfigurationClassEnhancer: 355 - @Bean
method
LocatorApplicationConfiguration.exclusiveLocatorApplicationBeanFactoryPostProcessor is
non-static and returns an object assignable to Spring's BeanFactoryPostProcessor
interface. This will result in a failure to process annotations such as @Autowired,
@Resource and @PostConstruct within the method's declaring @Configuration class. Add
the 'static' modifier to this method to avoid these container lifecycle issues; see
@Bean javadoc for complete details.
2019-09-01 11:02:49,919 INFO ode.distributed.internal.InternalLocator: 530 - Starting
peer location for Distribution Locator on 10.99.199.24[11235]
2019-09-01 11:02:49,925 INFO ode.distributed.internal.InternalLocator: 498 - Starting
Distribution Locator on 10.99.199.24[11235]
2019-09-01 11:02:49,926 INFO distributed.internal.tcpserver.TcpServer: 242 - Locator
was created at Sun Sep 01 11:02:49 PDT 2019
2019-09-01 11:02:49,927 INFO distributed.internal.tcpserver.TcpServer: 243 -
Listening on port 11235 bound on address 0.0.0.0/0.0.0.0
2019-09-01 11:02:49,928 INFO ternal.membership.gms.locator.GMSLocator: 162 - GemFire
peer location service starting. Other locators: localhost[10334] Locators preferred
as coordinators: true Network partition detection enabled: true View persistence
file: /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
docs/build/locator11235view.dat
2019-09-01 11:02:49,928 INFO ternal.membership.gms.locator.GMSLocator: 416 - Peer
locator attempting to recover from localhost/127.0.0.1:10334
2019-09-01 11:02:49,963 INFO ternal.membership.gms.locator.GMSLocator: 422 - Peer
locator recovered initial membership of
View[10.99.199.24(SpringLocatorOne:30043:locator)<ec><v0>:41000|0] members:
[10.99.199.24(SpringLocatorOne:30043:locator)<ec><v0>:41000]
2019-09-01 11:02:49,963 INFO ternal.membership.gms.locator.GMSLocator: 407 - Peer
locator recovered state from LocatorAddress
[socketInetAddress=localhost/127.0.0.1:10334, hostname=localhost, isIpString=false]
2019-09-01 11:02:49,965 INFO ode.distributed.internal.InternalLocator: 644 - Starting
distributed system

```

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership.

The ASF licenses this file to You under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Build-Date: 2019-04-19 11:49:13 -0700

Build-Id: onichols 0

Build-Java-Version: 1.8.0_192

Build-Platform: Mac OS X 10.14.4 x86_64

Product-Name: Apache Geode

Product-Version: 1.9.0

Source-Date: 2019-04-19 11:11:31 -0700

Source-Repository: release/1.9.0

Source-Revision: c0a73d1cb84986d432003bd12e70175520e63597

Native version: native code unavailable

Running on: 10.99.199.24/10.99.199.24, 8 cpu(s), x86_64 Mac OS X 10.13.6

Communications version: 100

Process ID: 30077

User: jblum

Current dir: /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build

Home dir: /Users/jblum

Command Line Parameters:

-ea

-Dspring.profiles.active=

-Dspring.data.gemfire.locator.name=SpringLocatorTwo

-Dspring.data.gemfire.locator.port=11235

-Dlogback.log.level=INFO

-javaagent:/Applications/IntelliJ IDEA 19

CE.app/Contents/lib/idea_rt.jar=51961:/Applications/IntelliJ IDEA 19

CE.app/Contents/bin

-Dfile.encoding=UTF-8

Class Path:

...

..

.

2019-09-01 11:02:54,112 INFO ode.distributed.internal.InternalLocator: 661 - Locator

```
started on 10.99.199.24[11235]
2019-09-01 11:02:54,113 INFO ode.distributed.internal.InternalLocator: 769 - Starting
server location for Distribution Locator on 10.99.199.24[11235]
2019-09-01 11:02:54,134 INFO nt.internal.locator.wan.LocatorDiscovery: 138 - Locator
discovery task exchanged locator information 10.99.199.24[11235] with
localhost[10334]: {-1=[10.99.199.24[10334]]}.
2019-09-01 11:02:54,242 INFO .SpringBootApacheGeodeLocatorApplication: 61 - Started
SpringBootApacheGeodeLocatorApplication in 6.137470354 seconds (JVM running for 6.667)
Press <enter> to exit!
```

Next, start up the second Locator app instance (you should see log output similar to above). Then, connect to the cluster of Locators using *Gfsh*:

Cluster of Locators

```
$ echo $GEMFIRE
/Users/jblum/pivdev/apache-geode-1.9.0

$ gfsh

-----
 /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /
 / /  _ _ /  _ _ /  _ _ _ _ /  _ _ _ _ /
 / /  _ _ /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /
 /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ 1.9.0
```

Monitor and Manage Apache Geode

```
gfsh>connect
Connecting to Locator at [host=localhost, port=10334] ..
Connecting to Manager at [host=10.99.199.24, port=1099] ..
Successfully connected to: [host=10.99.199.24, port=1099]

gfsh>list members

      Name                | Id
-----|-----
SpringLocatorOne | 10.99.199.24(SpringLocatorOne:30043:locator)<ec><v0>:41000
[Coordinator]
SpringLocatorTwo | 10.99.199.24(SpringLocatorTwo:30077:locator)<ec><v1>:41001
```

Using our `SpringBootApacheGeodeCacheServerApplication` main class from the previous section, we can configure and bootstrap an Apache Geode `CacheServer` application with Spring Boot and connect it to our cluster of Locators.

```
@SpringBootApplication
@CacheServerApplication(name = "SpringBootApacheGeodeCacheServerApplication")
@SuppressWarnings("unused")
public class SpringBootApacheGeodeCacheServerApplication {

    public static void main(String[] args) {

        new
        SpringApplicationBuilder(SpringBootApacheGeodeCacheServerApplication.class)
            .web(WebApplicationType.NONE)
            .build()
            .run(args);
    }

    @Configuration
    @UseLocators
    @Profile("clustered")
    static class ClusteredConfiguration { }

    @Configuration
    @EnableLocator
    @EnableManager(start = true)
    @Profile("!clustered")
    static class LonerConfiguration { }

}
```

Simply enable the "clustered" Profile by using a IDE Run Profile Configuration similar to:

```
-server -ea -Dspring.profiles.active=clustered -Dspring.data.gemfire.name=SpringServer
-Dspring.data.gemfire.cache.server.port=41414 -Dlogback.log.level=INFO
```

After the server starts up, you should see the new peer member in the cluster:

Cluster with Spring Boot configured and bootstrapped Apache Geode CacheServer

```
gfsh>list members
      Name      | Id
-----|-----
SpringLocatorOne | 10.99.199.24(SpringLocatorOne:30043:locator)<ec><v0>:41000
[Coordinator]
SpringLocatorTwo | 10.99.199.24(SpringLocatorTwo:30077:locator)<ec><v1>:41001
SpringServer     | 10.99.199.24(SpringServer:30216)<v2>:41002
```

Finally, we can even start additional Locators and Servers connected to this cluster using *Gfsh*:

```
gfsh>start locator --name=GfshLocator --port=12345 --log-level=config
Starting a Geode Locator in /Users/jblum/pivdev/lab/GfshLocator...
.....
Locator in /Users/jblum/pivdev/lab/GfshLocator on 10.99.199.24[12345] as GfshLocator
is currently online.
Process ID: 30259
Uptime: 5 seconds
Geode Version: 1.9.0
Java Version: 1.8.0_192
Log File: /Users/jblum/pivdev/lab/GfshLocator/GfshLocator.log
JVM Arguments: -Dgemfire.default.locators=10.99.199.24[11235],10.99.199.24[10334]
-Dgemfire.enable-cluster-configuration=true -Dgemfire.load-cluster-configuration-from
-dir=false -Dgemfire.log-level=config -Dgemfire.launcher.registerSignalHandlers=true
-Djava.awt.headless=true -Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /Users/jblum/pivdev/apache-geode-1.9.0/lib/geode-core-
1.9.0.jar:/Users/jblum/pivdev/apache-geode-1.9.0/lib/geode-dependencies.jar

gfsh>start server --name=GfshServer --server-port=45454 --log-level=config
Starting a Geode Server in /Users/jblum/pivdev/lab/GfshServer...
...
Server in /Users/jblum/pivdev/lab/GfshServer on 10.99.199.24[45454] as GfshServer is
currently online.
Process ID: 30295
Uptime: 2 seconds
Geode Version: 1.9.0
Java Version: 1.8.0_192
Log File: /Users/jblum/pivdev/lab/GfshServer/GfshServer.log
JVM Arguments:
-Dgemfire.default.locators=10.99.199.24[11235],10.99.199.24[12345],10.99.199.24[10334]
-Dgemfire.start-dev-rest-api=false -Dgemfire.use-cluster-configuration=true
-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/jblum/pivdev/apache-geode-1.9.0/lib/geode-core-
1.9.0.jar:/Users/jblum/pivdev/apache-geode-1.9.0/lib/geode-dependencies.jar

gfsh>list members
      Name      | Id
-----|-----
SpringLocatorOne | 10.99.199.24(SpringLocatorOne:30043:locator)<ec><v0>:41000
[Coordinator]
SpringLocatorTwo | 10.99.199.24(SpringLocatorTwo:30077:locator)<ec><v1>:41001
SpringServer     | 10.99.199.24(SpringServer:30216)<v2>:41002
GfshLocator      | 10.99.199.24(GfshLocator:30259:locator)<ec><v3>:41003
GfshServer       | 10.99.199.24(GfshServer:30295)<v4>:41004
```

You must be careful to vary the ports and name of your peer members appropriately. With Spring,

and Spring Boot for Apache Geode (SBDG) in particular, it really is that easy!

4.3. Building Manager Applications

As discussed in the previous sections above, it is possible to enable a Spring Boot configured and bootstrapped Apache Geode peer member node in the cluster to function as a *Manager*.

An Apache Geode *Manager* is a peer member node in the cluster running the Management Service, allowing the cluster to be managed and monitored using JMX based tools, like *Gfsh*, *JConsole* or *JVisualVM*, for instance. Any tool that uses the JMX API can connect to and manage an Apache Geode cluster for whatever purpose.

The cluster may have more than 1 *Manager* for redundancy. Only server-side, peer member nodes in the cluster may function as a *Manager*. Therefore, a `ClientCache` application cannot be a *Manager*.

To create a *Manager*, you use the SDG `@EnableManager` annotation.

The 3 primary uses of the `@EnableManager` annotation to create a *Manager* is:

1 - CacheServer Manager Application

```
@SpringBootApplication
@CacheServerApplication(name = "CacheServerManagerApplication")
@EnableManager(start = true)
class CacheServerManagerApplication {
    // ...
}
```

2 - Peer Cache Manager Application

```
@SpringBootApplication
@PeerCacheApplication(name = "PeerCacheManagerApplication")
@EnableManager(start = "true")
class SpringBootPeerCacheManagerApplication {
    // ...
}
```

3 - Locator Manager Application

```
@SpringBootApplication
@LocatorApplication(name = "LocatorManagerApplication")
@EnableManager(start = true)
class LocatorManagerApplication {
    // ...
}
```

#1 creates a peer `Cache` instance with a `CacheServer` component accepting client connections along with an embedded *Manager* enabling JMX clients to connect.

#2 creates only a peer `Cache` instance along with an embedded *Manager*. As a peer `Cache` with NO `CacheServer` component, clients are not able to connect to this node. It is merely a server managing data.

#3 creates a *Locator* instance with an embedded *Manager*.

In all configuration arrangements, the *Manager* was configured to start immediately.



See the `@EnableManager` annotation [Javadoc](#) for additional configuration options.

As of Apache Geode 1.11.0, you must now include additional Apache Geode dependencies on your Spring Boot application classpath to make your application a proper Apache Geode *Manager* in the cluster, particularly if you are also enabling the embedded HTTP service in the *Manager*.

The required dependencies are:

Additional Manager dependencies expressed in Gradle

```
runtime "org.apache.geode:geode-http-service"
runtime "org.apache.geode:geode-web"
runtime "org.springframework.boot:spring-boot-starter-jetty"
```

The embedded HTTP service (implemented with the Eclipse Jetty Servlet Container), runs the Management (Admin) REST API, which is used by tooling, such as *Gfsh*, to connect to the cluster over HTTP. In addition, it also runs the [Pulse](#) Monitoring Tool.

Even if you do not start the embedded HTTP service (Jetty Servlet Container), a *Manager* still requires the `geode-http-service`, `geode-web` and `spring-boot-starter-jetty` dependencies.

Chapter 5. Auto-configuration

The following Spring Framework, Spring Data for Apache Geode (SDG) and Spring Session for Apache Geode (SSDG) *Annotations* are implicitly declared by Spring Boot for Apache Geode's (SBDG) *Auto-configuration*.

- `@ClientCacheApplication`
- `@EnableGemfireCaching` (or alternatively, Spring Framework's `@EnableCaching`)
- `@EnableContinuousQueries`
- `@EnableGemfireFunctionExecutions`
- `@EnableGemfireFunctions`
- `@EnableGemfireRepositories`
- `@EnableLogging`
- `@EnablePdx`
- `@EnableSecurity`
- `@EnableSsl`
- `@EnableGemFireHttpSession`



This means you DO NOT need to explicitly declare any of these *Annotations* on your `@SpringBootApplication` class since they are provided by SBDG already. The only reason you would explicitly declare any of these *Annotations* is if you wanted to "override" Spring Boot's, and in particular, SBDG's *Auto-configuration*. Otherwise, it is unnecessary!



You should read the chapter in Spring Boot's Reference Documentation on [Auto-configuration](#).



You should review the chapter in Spring Data for Apache Geode's (SDG) Reference Documentation on [Annotation-based Configuration](#). For a quick reference, or an overview of Annotation-based Configuration, see [here](#).



Refer to the corresponding Sample [Guide](#) and [Code](#) to see Spring Boot Auto-configuration for Apache Geode in action!

5.1. Customizing Auto-configuration

You might ask how I can customize the *Auto-configuration* provided by SBDG if I do not explicitly declare the annotation?

For example, you may want to customize the member's "name". You know that the `@ClientCacheApplication` annotation provides the `name` attribute so you can set the client member's "name". But SBDG has already implicitly declared the `@ClientCacheApplication` annotation via *Auto-*

configuration on your behalf. What do you do?

Well, SBDG supplies a few very useful *Annotations* in this case.

For example, to set the (client or peer) member's name, you can use the `@UseMemberName` annotation, like so:

Setting the member's name using `@UseMemberName`

```
@SpringBootApplication
@UseMemberName("MyMemberName")
class SpringBootClientCacheApplication {
    ///...
}
```

Alternatively, you could set the `spring.application.name` or the `spring.data.gemfire.name` property in Spring Boot `application.properties`

Setting the member's name using the `spring.application.name` property

```
# Spring Boot application.properties

spring.application.name = MyMemberName
```

Or:

Setting the member's name using the `spring.data.gemfire.cache.name` property

```
# Spring Boot application.properties

spring.data.gemfire.cache.name = MyMemberName
```

In general, there are 3 ways to customize configuration, even in the context of SBDG's *Auto-configuration*:

1. Using [Annotations](#) provided by SBDG for common and popular concerns (e.g. naming client or peer members with `@UseMemberName`, or enabling durable clients with `@EnableDurableClient`).
2. Using well-known and documented [Properties](#) (e.g. `spring.application.name`, or `spring.data.gemfire.name`, or `spring.data.gemfire.cache.name`).
3. Using [Configurers](#) (e.g. `ClientCacheConfigurer`).



For the complete list of *documented* Properties, see [here](#).

5.2. Disabling Auto-configuration

Disabling Spring Boot *Auto-configuration* is [explained](#) in detail in Spring Boot's Reference Guide.

Disabling SBDG *Auto-configuration* was also [explained](#) in detail.

In a nutshell, if you want to disable any *Auto-configuration* provided by either Spring Boot or SBDG, then you can declare your intent in the `@SpringBootApplication` annotation, like so:

Disabling Specific Auto-configuration Classes

```
@SpringBootApplication(  
    exclude = { DataSourceAutoConfiguration.class, PdxAutoConfiguration.class }  
)  
class SpringBootClientCacheApplication {  
    // ...  
}
```



Make sure you understand what you are doing when you are "disabling" *Auto-configuration*.

5.3. Overriding Auto-configuration

Overriding SBDG *Auto-configuration* was [explained](#) in detail as well.

In a nutshell, if you want to override the default *Auto-configuration* provided by SBDG then you must annotate your `@SpringBootApplication` class with your intent.

For example, say you want to configure and bootstrap an Apache Geode `CacheServer` application (a peer; not a client), then you would:

Overriding the default `ClientCache` Auto-Configuration by configuring & bootstrapping a `CacheServer` application

```
@SpringBootApplication  
@CacheServerApplication  
class SpringBootCacheServerApplication {  
    // ...  
}
```

Even when you explicitly declare the `@ClientCacheApplication` annotation on your `@SpringBootApplication` class, like so:

Overriding by explicitly declaring `@ClientCacheApplication`

```
@SpringBootApplication  
@ClientCacheApplication  
class SpringBootClientCacheApplication {  
    // ...  
}
```

You are overriding SBDG's *Auto-configuration* of the `ClientCache` instance. As a result, you now have also implicitly consented to being responsible for other aspects of the configuration (e.g. *Security*)! Why?

This is because in certain cases, like *Security*, certain aspects of *Security* configuration (e.g. SSL) must be configured before the cache instance is created. And, Spring Boot always applies user configuration before *Auto-configuration* partially to determine what needs to be auto-configured in the first place.



Especially make sure you understand what you are doing when you are "overriding" *Auto-configuration*.

5.4. Replacing Auto-configuration

We will simply refer you to the Spring Boot Reference Guide on replacing *Auto-configuration*. See [here](#).

5.5. Auto-configuration Explained

This section covers the SBDG provided *Auto-configuration* classes corresponding to the SDG *Annotations* in more detail.

To review the complete list of SBDG *Auto-configuration* classes, [see here](#).

5.5.1. @ClientCacheApplication



The `ClientCacheAutoConfiguration` class corresponds to the `@ClientCacheApplication` annotation.

SBDG [starts](#) with the opinion that application developers will primarily be building Apache Geode [client applications](#) using Spring Boot.

Technically, this means building Spring Boot applications with an Apache Geode `ClientCache` instance connected to a dedicated cluster of Apache Geode servers that manage the data as part of a [client/server](#) topology.

By way of example, this means you **do not** need to explicitly declare and annotate your `@SpringBootApplication` class with SDG's `@ClientCacheApplication` annotation, like so:

Do Not Do This

```
@SpringBootApplication
@ClientCacheApplication
class SpringBootClientCacheApplication {
    // ...
}
```

This is because SBDG's provided *Auto-configuration* class is already meta-annotated with SDG's `@ClientCacheApplication` annotation. Therefore, you simply need:

Do This

```
@SpringBootApplication
class SpringBootClientCacheApplication {
    // ...
}
```



Refer to SDG's Reference Documentation for more details on Apache Geode [cache applications](#), and [client/server applications](#) in particular.

5.5.2. @EnableGemfireCaching



The `CachingProviderAutoConfiguration` class corresponds to the `@EnableGemfireCaching` annotation.

If you simply used the core Spring Framework to configure Apache Geode as a *caching provider* in [Spring's Cache Abstraction](#), you would need to do this:

Configuring caching using the Spring Framework

```
@SpringBootApplication
@EnableCaching
class CachingUsingApacheGeodeConfiguration {

    @Bean
    GemfireCacheManager cacheManager(GemFireCache cache) {

        GemfireCacheManager cacheManager = new GemfireCacheManager();

        cacheManager.setCache(cache);

        return cacheManager;
    }
}
```

If you were using Spring Data for Apache Geode's `@EnableGemfireCaching` annotation, then the above configuration could be simplified to:

Configuring caching using Spring Data Geode

```
@SpringBootApplication
@EnableGemfireCaching
class CachingUsingApacheGeodeConfiguration {

}
```

And, if you use SBDG, then you only need to do this:

```
@SpringBootApplication
class CachingUsingApacheGeodeConfiguration {

}
```

This allows you to focus on the areas in your application that would benefit from caching without having to enable the plumbing. Simply demarcate the service methods in your application that are good candidates for caching:

Using caching in your application

```
@Service
class CustomerService {

    @Caching("CustomersByName")
    Customer findBy(String name) {
        // ...
    }
}
```



Refer to the [documentation](#) for more details.

5.5.3. @EnableContinuousQueries



The `ContinuousQueryAutoConfiguration` class corresponds to the `@EnableContinuousQueries` annotation.

Without having to enable anything, you simply annotate your application (POJO) component method(s) with the SDG `@ContinuousQuery` annotation to register a CQ and start receiving events. The method acts as a `CqEvent` handler, or in Apache Geode's case, the method would be an implementation of `CqListener`.

Declare application CQs

```
@Component
class MyCustomerApplicationContinuousQueries {

    @ContinuousQuery("SELECT customer.* FROM /Customers customers"
        + " WHERE customer.getSentiment().name().equalsIgnoreCase('UNHAPPY')")
    public void handleUnhappyCustomers(CqEvent event) {
        // ...
    }
}
```

As shown above, you define the events you are interested in receiving by using a OQL query with a finely tuned query predicate describing the events of interests and implement the handler method

to process the events (e.g. apply a credit to the customer's account and follow up in email).



Refer to the [documentation](#) for more details.

5.5.4. `@EnableGemfireFunctionExecutions` & `@EnableGemfireFunctions`



The `FunctionExecutionAutoConfiguration` class corresponds to both the `@EnableGemfireFunctionExecutions` and `@EnableGemfireFunctions` annotations.

Whether you need to *execute* a `Function` or *implement* a `Function`, SBDG will detect the Function definition and auto-configure it appropriately for use in your Spring Boot application. You only need to define the Function execution or implementation in a package below the main `@SpringBootApplication` class.

Declare a Function Execution

```
package example.app.functions;

@OnRegion("Accounts")
interface MyCustomerApplicationFunctions {

    void applyCredit(Customer customer);

}
```

Then you can inject the Function execution into any application component and use it:

Use the Function

```
package example.app.service;

@Service
class CustomerService {

    @Autowired
    private MyCustomerApplicationFunctions customerFunctions;

    void analyzeCustomerSentiment(Customer customer) {

        // ...

        this.customerFunctions.applyCredit(customer);

        // ...

    }

}
```

The same pattern basically applies to Function implementations, except in the implementation case, SBDG "registers" the Function implementation for use (i.e. to be called by a Function

execution).

The point is, you are simply focusing on defining the logic required by your application, and not worrying about how Functions are registered, called, etc. SBDG is handling this concern for you!



Function implementations are typically defined and registered on the server-side.



Refer to the [documentation](#) for more details.

5.5.5. @EnableGemfireRepositories



The `GemFireRepositoriesAutoConfigurationRegistrar` class corresponds to the `@EnableGemfireRepositories` annotation.

Like Functions, you are only concerned with the data access operations (e.g. basic CRUD and simple Queries) required by your application to carry out its functions, not how to create and perform them (e.g. `Region.get(key)` & `Region.put(key, obj)`) or execute (e.g. `Query.execute(arguments)`).

Simply define your Spring Data Repository:

Define an application-specific Repository

```
package example.app.repo;

interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findBySentimentEqualTo(Sentiment sentiment);

}
```

And use it:

```
package example.app.service;

@Service
class CustomerService {

    @Autowired
    private CustomerRepository repository;

    public void processCustomersWithSentiment(Sentiment sentiment) {

        this.repository.findBySentimentEqualTo(sentiment).forEach(customer -> { /* ... */
    });

    // ...
}
}
```

Your application-specific *Repository* simply needs to be declared in a package below the main `@SpringBootApplication` class. Again, you are only focusing on the data access operations and queries required to carry out the functions of your application, nothing more.



Refer to the [documentation](#) for more details.

5.5.6. @EnableLogging



The `LoggingAutoConfiguration` class corresponds to the `@EnableLogging` annotation.

Logging is an essential application concern to understand what is happening in the system along with when and where the event occurred. As such, SBDG auto-configures logging for Apache Geode by default, using the default log-level, "config".

If you wish to change an aspect of logging, such as the log-level, you would typically do this in Spring Boot `application.properties`:

Change the log-level for Apache Geode

```
# Spring Boot application.properties.

spring.data.gemfire.cache.log-level=debug
```

Other aspects may be configured as well, such as the log file size and disk space limits for the file system location used to store the Apache Geode log files at runtime.

Under-the-hood, Apache Geode's logging is based on Log4j. Therefore, you can configure Apache Geode logging using any logging provider (e.g. Logback) and configuration metadata appropriate for that logging provider so long as you supply the necessary adapter between Log4j and whatever

logging system you are using. For instance, if you include `org.springframework.boot:spring-boot-starter-logging` then you will be using Logback and you will need the `org.apache.logging.log4j:log4j-to-slf4j` adapter.

5.5.7. @EnablePdx



The `PdxSerializationAutoConfiguration` class corresponds to the `@EnablePdx` annotation.

Anytime you need to send an object over the network, overflow or persist an object to disk, then your application domain object must be *serializable*. It would be painful to have to implement `java.io.Serializable` in everyone of your application domain objects (e.g. `Customer`) that would potentially need to be serialized.

Furthermore, using *Java Serialization* may not be ideal (e.g. the most portable or efficient) in all cases, or even possible in other cases (e.g. when you are using a 3rd party library for which you have no control over).

In these situations, you need to be able to send your object anywhere without unduly requiring the class type to be serializable as well as to exist on the classpath for every place it is sent. Indeed, the final destination may not even be a Java application! This is where Apache Geode [PDX Serialization](#) steps into help.

However, you don't have to figure out how to configure PDX to identify the application class types that will need to be serialized. You simply define your class type:

Customer class

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

    @Id
    private Long id;

    @Indexed
    private String name;

    // ...
}
```

And, SBDG's *Auto-configuration* will handle the rest!



Refer to the [documentation](#) for more details.

5.5.8. @EnableSecurity



The `ClientSecurityAutoConfiguration` class and `PeerSecurityAutoConfiguration` class corresponds to the `@EnableSecurity` annotation, but applies Security, and specifically, Authentication/Authorization configuration for both clients and servers.

Configuring your Spring Boot, Apache Geode `ClientCache` application to properly authenticate with a cluster of secure Apache Geode servers is as simple as setting a *username* and *password* in Spring Boot `application.properties`:

Supplying Authentication Credentials

```
# Spring Boot application.properties

spring.data.gemfire.security.username=Batman
spring.data.gemfire.security.password=r0b!n5ucks
```



Authentication is even easier to configure in a managed environment like PCF when using PCC; you don't have to do anything!

Authorization is configured on the server-side and is made simple with SBDG and the help of [Apache Shiro](#). Of course, this assumes you are using SBDG to configure and bootstrap your Apache Geode cluster in the first place, which is [possible](#), and made even easier with SBDG.



Refer to the [documentation](#) for more details.

5.5.9. `@EnableSsl`



The `SslAutoConfiguration` class corresponds to the `@EnableSsl` annotation.

Configuring SSL for secure transport (TLS) between your Spring Boot, Apache Geode `ClientCache` application and the cluster can be a real problematic task, especially to get correct from the start. So, it is something that SBDG makes simple to do out-of-the-box.

Simply supply a `trusted.keystore` file containing the certificates in a well-known location (e.g. root of your application classpath) and SBDG's *Auto-configuration* will kick in and handle of the rest.

This is useful during development, but we highly recommend using a more secure procedure (e.g. integrating with a secure credential store like LDAP, CredHub or Vault) when deploying your Spring Boot application to production.



Refer to the [documentation](#) for more details.

5.5.10. `@EnableGemFireHttpSession`



The `SpringSessionAutoConfiguration` class corresponds to the `@EnableSsl` annotation.

Configuring Apache Geode to serve as the (HTTP) Session state caching provider using Spring Session is as simple as including the correct starter, e.g. `spring-geode-starter-session`.

Using Spring Session

```
<dependency>
  <groupId>org.springframework.geode</groupId>
  <artifactId>spring-geode-starter-session</artifactId>
  <version>1.5.0-RC1</version>
</dependency>
```

With Spring Session, and specifically Spring Session for Apache Geode (SSDG), on the classpath of your Spring Boot, Apache Geode `ClientCache` Web application, you can manage your (HTTP) Session state with Apache Geode. No further configuration is needed. SBDG *Auto-configuration* detects Spring Session on the application classpath and does the right thing.



Refer to the [documentation](#) for more details.

5.5.11. RegionTemplateAutoConfiguration

The SBDG `RegionTemplateAutoConfiguration` class has no corresponding SDG *Annotation*. However, the *Auto-configuration* of a `GemfireTemplate` for every single Apache Geode `Region` defined and declared in your Spring Boot application is supplied by SBDG never-the-less.

For example, if you defined a Region using:

Region definition using JavaConfig

```
@Configuration
class GeodeConfiguration {

    @Bean("Customers")
    ClientRegionFactoryBean<Long, Customer> customersRegion(GemFireCache cache) {

        ClientRegionFactoryBean<Long, Customer> customersRegion =
            new ClientRegionFactoryBean<>();

        customersRegion.setCache(cache);
        customersRegion.setShortcut(ClientRegionShortcut.PROXY);

        return customersRegion;
    }
}
```

Alternatively, you could define the "Customers" Region using:

Region definition using `@EnableEntityDefinedRegions`

```
@Configuration
@EnableEntityDefinedRegion(basePackageClasses = Customer.class)
class GeodeConfiguration {

}
```

Then, SBDG will supply a `GemfireTemplate` instance that you can use to perform low-level, data access operations (indirectly) on the "Customers" Region:

Use the `GemfireTemplate` to access the "Customers" Region

```
@Repository
class CustomersDao {

    @Autowired
    @Qualifier("customersTemplate")
    private GemfireTemplate customersTemplate;

    Customer findById(Long id) {
        return this.customerTemplate.get(id);
    }
}
```

You do not need to explicitly configure `GemfireTemplates` for each Region you need to have low-level data access to (e.g. such as when you are not using the Spring Data Repository abstraction).

Be careful to "qualify" the `GemfireTemplate` for the Region you need data access to, especially given that you will probably have more than 1 Region defined in your Spring Boot application.



Refer to the [documentation](#) for more details.

Chapter 6. Declarative Configuration

The primary purpose of any software development framework is to help you be *productive* as *quickly* and as *easily* as possible, and to do so in a *reliable* manner.

As application developers, we want a framework to provide constructs that are both intuitive and familiar so that their behaviors are boringly predictable. This provided convenience not only helps you hit the ground running in the right direction sooner but increases your focus on the application domain so you are able to better understand the problem you are trying to solve in the first place. Once the problem domain is well understood, you are more apt to make informed decisions about the design, which leads to better outcomes, faster.

This is exactly what Spring Boot's *auto-configuration* provides for you... enabling features, services and supporting infrastructure for Spring applications in a loosely integrated way by using conventions (e.g. classpath) that ultimately helps you keep your attention and focus on solving the problem at hand and not on the plumbing.

For example, if you are building a Web application, simply include the `org.springframework.boot:spring-boot-starter-web` dependency on your application classpath. Not only will Spring Boot enable you to build Spring Web MVC Controllers appropriate to your application UC (your responsibility), but will also bootstrap your Web app in an embedded Servlet Container on startup (Boot's responsibility).

This saves you from having to handle many low-level, repetitive and tedious development tasks that are highly error-prone when you are simply trying to solve problems. You don't have to care how the plumbing works until you do. And, when you do, you will be better informed and prepared to do so.

It is also equally essential that frameworks, like Spring Boot, get out of the way quickly when application requirements diverge from the provided defaults. The is the beautiful and powerful thing about Spring Boot and why it is second to none in its class.

Still, *auto-configuration* does not solve every problem all the time. Therefore, you will need to use declarative configuration in some cases, whether expressed as bean definitions, in properties or by some other means. This is so frameworks don't leave things to chance, especially when they are ambiguous. The framework simply gives you a choice.

Now, that we explained the motivation behind this chapter, let's outline what we will discuss:

- Refer you to the SDG *Annotations* covered by SBDG's *Auto-configuration*
- List all SDG *Annotations* not covered by SBDG's *Auto-configuration*
- Cover the SBDG, SSDG and SDG *Annotations* that must be declared explicitly and that provide the most value and productivity when getting started using either Apache Geode in Spring [Boot] applications.



SDG refers to [Spring Data for Apache Geode](#). SSDG refers to [Spring Session for Apache Geode](#) and SBDG refers to *Spring Boot for Apache Geode*, this project.



The list of *SDG Annotations* covered by SBDG's *Auto-configuration* is discussed in detail in the [Appendix](#), in the section, [Auto-configuration vs. Annotation-based configuration](#).

To be absolutely clear about which *SDG Annotations* we are referring to, we mean the *SDG Annotations* in the package: org.springframework.data.gemfire.config.annotation.

Additionally, in subsequent sections, we will cover which *Annotations* are added by SBDG.

6.1. Auto-configuration

Auto-configuration was explained in complete detail in the chapter, "[Auto-configuration](#)".

6.2. Annotations not covered by Auto-configuration

The following *SDG Annotations* are not implicitly applied by SBDG's *Auto-configuration*:

- `@EnableAutoRegionLookup`
- `@EnableBeanFactoryLocator`
- `@EnableCacheServer(s)`
- `@EnableCachingDefinedRegions`
- `@EnableClusterConfiguration`
- `@EnableClusterDefinedRegions`
- `@EnableCompression`
- `@EnableDiskStore(s)`
- `@EnableEntityDefinedRegions`
- `@EnableEviction`
- `@EnableExpiration`
- `@EnableGatewayReceiver`
- `@EnableGatewaySender(s)`
- `@EnableGemFireAsLastResource`
- `@EnableGemFireMockObjects`
- `@EnableHttpService`
- `@EnableIndexing`
- `@EnableOffHeap`
- `@EnableLocator`
- `@EnableManager`
- `@EnableMemcachedServer`
- `@EnablePool(s)`

- `@EnableRedisServer`
- `@EnableStatistics`
- `@UseGemFireProperties`



This was also covered [here](#).

Part of the reason for this is because several of the *Annotations* are server-specific:

- `@EnableCacheServer(s)`
- `@EnableGatewayReceiver`
- `@EnableGatewaySender(s)`.
- `@EnableHttpService`
- `@EnableLocator`
- `@EnableManager`
- `@EnableMemcachedServer`
- `@EnableRedisServer`

And, we [already stated](#) that SBDG is opinionated about providing a `ClientCache` instance out-of-the-box.

Other *Annotations* are driven by need, for example:

- `@EnableAutoRegionLookup` & `@EnableBeanFactoryLocator` - really only useful when mixing configuration metadata formats, e.g. Spring config with Apache Geode `cache.xml`. This is usually only the case if you have legacy `cache.xml` config to begin with, otherwise, don't do this!
- `@EnableCompression` - requires the Snappy Compression Library on your application classpath.
- `@EnableDiskStore(s)` - only used for overflow and persistence.
- `@EnableOffHeap` - enables data to be stored in main memory, which is only useful when your application data (i.e. Objects stored in Apache Geode) are generally uniform in size.
- `@EnableGemFireAsLastResource` - only needed in the context of JTA Transactions.
- `@EnableStatistics` - useful if you need runtime metrics, however enabling statistics gathering does consume considerable system resources (e.g. CPU & Memory).

While still other *Annotations* require more careful planning, for example:

- `@EnableEviction`
- `@EnableExpiration`
- `@EnableIndexing`

One in particular is used exclusively for Unit Testing:

- `@EnableGemFireMockObjects`

The bottom-line is, a framework should not *Auto-configure* every possible feature, especially when

the features consume additional system resources, or requires more careful planning as determined by the use case.

Still, all of these *Annotations* are available for the application developer to use when needed.

6.3. Productivity Annotations

This section calls out the *Annotations* we believe to be most beneficial for your application development purposes when using Apache Geode in Spring Boot applications.

6.3.1. `@EnableClusterAware` (SBDG)

The `@EnableClusterAware` annotation is arguably the most powerful and valuable *Annotation* in the set of *Annotations*!

When you annotate your main `@SpringBootApplication` class with `@EnableClusterAware`:

Declaring `@EnableClusterAware`

```
@SpringBootApplication
@EnableClusterAware
class SpringBootApacheGeodeClientCacheApplication { }
```

Your Spring Boot, Apache Geode `ClientCache` application is able to seamlessly switch between client/server and local-only topologies with no code or configuration changes, regardless of the runtime environment (e.g. local/standalone vs. cloud-managed environments).

When a cluster of Apache Geode servers is detected, the client application will send and receive data to and from the cluster. If a cluster is not available, then the client automatically switches to storing data locally on the client using `LOCAL` Regions.

Additionally, the `@EnableClusterAware` annotation is meta-annotated with SDG's `@EnableClusterConfiguration` annotation.

The `@EnableClusterConfiguration` enables configuration metadata defined on the client (e.g. Region and Index definitions) as needed by the application based on requirements and use cases, to be sent to the cluster of servers. If those schema objects are not already present, they will be created by the servers in the cluster in such a way that the servers will remember the configuration on restart as well as provide the configuration to new servers joining the cluster when scaling out. This feature is careful not to stomp on any existing Region or Index objects already present on the servers, particularly since you may already have data stored in the Regions.

The primary motivation behind the `@EnableClusterAware` annotation is to allow you to switch environments with very little effort. It is a very common development practice to debug and test your application locally, in your IDE, then push up to a production-like environment for more rigorous integration testing.

By default, the configuration metadata is sent to the cluster using a non-secure HTTP connection. Using HTTPS, changing host and port, and configuring the data management policy used by the

servers when creating Regions is all configurable.



Refer to the section in the SDG Reference Guide on [Configuring Cluster Configuration Push](#) for more details.

@EnableClusterAware, strictMatch

The `strictMatch` attribute has been added to the `@EnableClusterAware` annotation to enable *fail-fast* behavior. `strictMatch` is set to `false` by default.

Essentially, when you set `strictMatch` to `true`, then you are saying that your Spring Boot, Apache Geode `ClientCache` application requires an Apache Geode cluster to exist, i.e. the application requires a client/server topology to operate and that the application should fail to start and run if a cluster is not present. The application should not startup in a LOCAL-only capacity.

When `strictMatch` is set to `true` and an Apache Geode cluster is not present, then your Spring Boot, Apache Geode `ClientCache` application will fail to start with a `ClusterNotFoundException`. The application will not attempt to startup in a LOCAL-only capacity.

You can explicitly set the `strictMatch` attribute programmatically using the `@EnableClusterAware` annotation:

Set `@EnableClusterAware.strictMatch`

```
@SpringBootApplication
@EnableClusterAware(strictMatch = true)
class SpringBootApacheGeodeClientCacheApplication { }
```

Alternatively, you can set `strictMatch` using the corresponding property in Spring Boot `application.properties`:

Set `strictMatch` using a property

```
# Spring Boot application.properties

spring.boot.data.gemfire.cluster.condition.match.strict=true
```

This is convenient when you need to apply this configuration setting conditionally based on a Spring Profile.

When you adjust the *log-level* of the `org.springframework.geode.config.annotation.ClusterAwareConfiguration` *Logger* to `INFO`, then you will get more details from the `@EnableClusterAware` functionality when applying the logic to determine the presence of an Apache Geode cluster, such as which explicitly or implicitly configured connections were successful.

For example:

@EnableClusterAware INFO log output

```
2021-01-20 14:02:28,740 INFO fig.annotation.ClusterAwareConfiguration: 476 - Failed
to connect to localhost[40404]
2021-01-20 14:02:28,745 INFO fig.annotation.ClusterAwareConfiguration: 476 - Failed
to connect to localhost[10334]
2021-01-20 14:02:28,746 INFO fig.annotation.ClusterAwareConfiguration: 470 -
Successfully connected to localhost[57649]
2021-01-20 14:02:28,746 INFO fig.annotation.ClusterAwareConfiguration: 576 - Cluster
was found; Auto-configuration made [1] successful connection(s);
2021-01-20 14:02:28,746 INFO fig.annotation.ClusterAwareConfiguration: 586 - Spring
Boot application is running in a client/server topology, using a standalone Apache
Geode-based cluster
```



An attempt is always made to connect to `localhost` on the default *Locator* port, `10334`, as well as the default *CacheServer* port, `40404`.



You can force a successful match always by setting the `spring.boot.data.gemfire.cluster.condition.match` property to `true` in Spring Boot `application.properties`. This is sometimes useful for testing purposes.

6.3.2. @EnableCachingDefinedRegions, @EnableClusterDefinedRegions & @EnableEntityDefinedRegions (SDG)

These *Annotations* are used to create Regions in the cache to manage your application data.

Of course, you can create Regions using Java configuration and the Spring API as follows:

Creating a Region with Spring JavaConfig

```
@Configuration
class GeodeConfiguration {

    @Bean("Customers")
    ClientRegionFactoryBean<Long, Customer> customersRegion(GemFireCache cache) {

        ClientRegionFactoryBean<Long, Customer> customers = new
        ClientRegionFactoryBean<>();

        customers.setCache(cache);
        customers.setShortcut(ClientRegionShortcut.PROXY);

        return customers;
    }
}
```

Or XML:

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

However, using the provided Annotations is far easier, especially during development when the complete Region configuration may be unknown and you simply want to create a Region to persist your application data and move on.

`@EnableCachingDefinedRegions`

The `@EnableCachingDefinedRegions` annotation is used when you have application components registered in the Spring Container that are annotated with Spring or JSR-107, JCache [annotations](#).

Caches that identified by name in the caching annotations are used to create Regions holding the data you want cached.

For example, given:

Defining Regions based on Spring or JSR-107 JCache Annotations

```
@Service
class CustomerService {

    @Cacheable(cacheNames = "CustomersByAccountNumber", key = "#account.number")
    Customer findBy(Account account) {
        // ...
    }
}
```

When your main `@SpringBootApplication` class is annotated with `@EnableCachingDefinedRegions`:

Using `@EnableCachingDefinedRegions`

```
@SpringBootApplication
@EnableCachingDefineRegions
class SpringBootApacheGeodeClientCacheApplication { }
```

Then, SBDG would create a client `PROXY` Region (or `PARTITION_REGION` if your application were a peer member of the cluster) with the name `"CustomersByAccountNumber"` as if you created the Region using either the JavaConfig or XML approaches shown above.

You can use the `clientRegionShortcut` or `serverRegionShortcut` attribute to change the data management policy of the Regions created on the client or servers, respectively.

For client Regions, you can additionally assign a specific Pool of connections used by the client `*PROXY` Regions to send data to the cluster by setting the `poolName` attribute.

@EnableEntityDefinedRegions

Like `@EnableCachingDefinedRegions`, `@EnableEntityDefinedRegions` allows you to create Regions based on the entity classes you have defined in your application domain model.

For instance, if you have entity class annotated with SDG's `@Region` mapping annotation:

Customer entity class annotated with `@Region`

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

    @Id
    private Long id;

    @Indexed
    private String name;

}
```

Then SBDG will create Regions from the name specified in the `@Region` mapping annotation on the entity class. In this case, the `Customer` application-defined entity class will result in the creation of a Region named "Customers" when the main `@SpringBootApplication` class is annotated with `@EnableEntityDefinedRegions`:

Using `@EnableEntityDefinedRegions`

```
@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class,
    clientRegionShortcut = ClientRegionShortcut.CACHING_PROXY)
class SpringBootApacheGeodeClientCacheApplication { }
```

Like the `@EnableCachingDefinedRegions` annotation, you can set the client and server Region data management policy using the `clientRegionShortcut` and `serverRegionShortcut` attributes, respectively, as well as set a dedicated Pool of connections used by client Regions with the `poolName` attribute.

However, unlike the `@EnableCachingDefinedRegions` annotation, users are required to specify either the `basePackage`, or the type-safe alternative, `basePackageClasses` attribute (recommended) when using the `@EnableEntityDefinedRegions` annotation.

Part of the reason for this is that `@EnableEntityDefinedRegions` performs a component scan for the entity classes defined by your application. The component scan loads each class to inspect the *Annotation* metadata for that class. This is not unlike the JPA entity scan when working with JPA providers like Hibernate.

Therefore, it is customary to limit the scope of the scan, otherwise you end up potentially loading many classes unnecessarily so. After all, the JVM uses dynamic linking to only load classes when needed.

Both the `basePackages` and `basePackageClasses` attributes accept an array of values. With `basePackageClasses` you only need to refer to a single class type in that package and every class in that package as well as classes in the sub-packages will be scanned to determine if the class type represents an entity. A class type is an entity if it is annotated with the `@Region` mapping annotation, otherwise it is not considered an entity.

By example, suppose you had the following structure:

Entity Scan

```
- example.app.crm.model
  |- Customer.class
  |- NonEntity.class
  |- contact
    |- Address.class
    |- PhoneNumber.class
    |- AnotherNonEntity.class
- example.app.accounts.model
  |- Account.class
...
..
.
```

Then, you could configure the `@EnableEntityDefinedRegions` as follows:

Targeting with `@EnableEntityDefinedRegions`

```
@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = { NonEntity.class, Account.class } )
class SpringBootApacheGeodeClientCacheApplication { }
```

If `Customer`, `Address`, `PhoneNumber` and `Account` were all entity classes properly annotated with `@Region`, then the component scan would pick up all these classes and create Regions for them. The `NonEntity` class only serves as a marker in this case pointing to where (i.e. what package) the scan should begin.

Additionally, the `@EnableEntityDefinedRegions` annotation provides *include* and *exclude* filters, the same as the core Spring Frameworks `@ComponentScan` annotation.



Refer to the SDG Reference Guide on [Configuring Regions](#) for more details.

`@EnableClusterDefinedRegions`

Sometimes it is ideal or even necessary to pull configuration from the cluster (rather than push to the cluster). That is, you want the Regions defined on the servers to be created on the client and used by your application.

This is as simple as annotating your main `@SpringBootApplication` class with `@EnableClusterDefinedRegions`:

Using `@EnableClusterDefinedRegions`

```
@SpringBootApplication
@EnableClusterDefinedRegions
class SpringBootApacheGeodeClientCacheApplication { }
```

Every Region that exists on the cluster of servers will have a corresponding `PROXY` Region defined and created on the client as a bean in your Spring Boot application.

If the cluster of servers defines a Region called `"ServerRegion"` you can inject the client `PROXY` Region by the same name (i.e. `"ServerRegion"`) into your Spring Boot application and use it:

Using a server-side Region on the client

```
@Component
class SomeApplicationComponent {

    @Resource(name = "ServerRegion")
    private Region<Integer, EntityType> serverRegion;

    public void someMethod() {

        EntityType entity = new EntityType();

        this.serverRegion.put(1, entity);

        // ...
    }
}
```

Of course, SBDG *auto-configures* a `GemfireTemplate` for the `"ServerRegion"` Region (as described [here](#)), so a better way to interact with the client `PROXY` Region corresponding to the `"ServerRegion"` Region on the server is to inject the template:

```
@Component
class SomeApplicationComponent {

    @Autowired
    @Qualifier("serverRegionTemplate")
    private GemfireTemplate serverRegionTemplate;

    public void someMethod() {

        EntityType entity = new EntityType();

        this.serverRegionTemplate.put(1, entity);

        //...
    }
}
```



Refer to the SDG Reference Guide on [Configuring Cluster-defined Regions](#) for more details.

6.3.3. @EnableIndexing (SDG)

Only when using `@EnableEntityDefinedRegions` can you also use the `@EnableIndexing` annotation. This is because `@EnableIndexing` requires the entities to be scanned and analyzed for mapping metadata defined on the class type of the entity. This includes annotations like Spring Data Commons `@Id` annotation as well as SDG provided annotations, `@Indexed` and `@LuceneIndexed`.

The `@Id` annotation identifies the (primary) key of the entity. The `@Indexed` defines OQL Indexes on object fields which are used in the predicates of your OQL Queries. The `@LuceneIndexed` annotation is used to define Apache Lucene Indexes required for searches.



Lucene Indexes can only be created on **PARTITION** Regions, and **PARTITION** Regions are only defined on the server-side.

You may have noticed that the `Customer` entity class's `name` field was annotated with `@Indexed`.

Customer entity class with `@Indexed` annotated `name` field

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

    @Id
    private Long id;

    @Indexed
    private String name;

}
```

As a result, when our main `@SpringBootApplication` class is annotated with `@EnableIndexing`:

Using `@EnableIndexing`

```
@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableIndexing
class SpringBootApacheGeodeClientCacheApplication { }
```

An Apache Geode OQL Index for the `Customer.name` field will be created thereby making OQL Queries on Customers by name use this Index.



Keep in mind that OQL Indexes are not persistent between restarts (i.e. Apache Geode maintains Indexes in-memory only). An OQL Index is always rebuilt when the node is restarted.

When you combine `@EnableIndexing` with either `@EnableClusterConfiguration` or `@EnableClusterAware`, then the Index definitions will be pushed to the server-side Regions where OQL Queries are generally executed.



Refer to the SDG Reference Guide on [Configuring Indexes](#) for more details.

6.3.4. `@EnableExpiration` (SDG)

It is often useful to define both *Eviction* and *Expiration* policies, particularly with a system like Apache Geode, especially given it primarily keeps data in-memory, on the JVM Heap. As you can imagine your data volume size may far exceed the amount of available JVM Heap memory and/or keeping too much data on the JVM Heap can cause Garbage Collection (GC) issues.



You can enable off-heap (or main memory usage) capabilities by declaring SDG's `@EnableOffHeap` annotation. Refer to the SDG Reference Guide on [Configuring Off-Heap Memory](#) for more details.

Defining *Eviction* and *Expiration* policies is a useful for limiting what is kept in memory and for how long.

While [configuring Eviction](#) is easy with SDG, we particularly want to call out *Expiration* since [configuring Expiration](#) has special support in SDG.

With SDG, it is possible to define the *Expiration* policies associated with a particular application class type on the class type itself, using the `@Expiration`, `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` annotations.



Refer to the Apache Geode [User Guide](#) for more details on the different Expiration Types (i.e. *Idle Timeout* (TTI) vs. *Time-To-Live* (TTL)).

For example, suppose we want to limit the number of `Customers` maintained in memory for a period of time (measured in seconds) based on the last time a `Customer` was accessed (e.g. *read*). We can then define an *Idle Timeout* Expiration policy on our `Customer` class type, like so:

Customer entity class with `@Indexed` annotated `name` field

```
@Region("Customers")
@IdleTimeoutExpiration(action = "INVALIDATE", timeout = "300")
class Customer {

    @Id
    private Long id;

    @Indexed
    private String name;

}
```

The `Customer` entry in the "Customers" Region will be *invalidated* after **300 seconds** (or **5 minutes**).

All we need to do to enable annotation-based Expiration policies is annotate our main `@SpringBootApplication` class with `@EnableExpiration`:

Enabling Expiration

```
@SpringBootApplication
@EnableExpiration
class SpringBootApacheGeodeApplication { }
```



Technically, this entity class specific Annotation-based Expiration policy is implemented using Apache Geode's `CustomExpiry` interface.



Refer to the SDG Reference Guide for more details on [configuring Expiration](#), along with [Annotation-based Data Expiration](#) in particular.

6.3.5. `@EnableGemFireMockObjects` (STDG)

Software Testing in general, and *Unit Testing* in particular, are a very important development tasks

to ensure the quality of your Spring Boot applications.

Apache Geode can make testing difficult in some cases, especially when tests have to be written as *Integration Tests* in order to assert the correct behavior. This can be very costly and lengthens the feedback cycle. Fortunately, it is possible to write *Unit Tests* as well!

Spring has your back and once again provides a framework for testing Spring Boot applications using Apache Geode. This is where the [Spring Test for Apache Geode \(STDG\)](#) project can help, particularly with *Unit Testing*.

For example, if you do not care what Apache Geode would actually do in certain cases and only care about the "contract", which is what mocking a collaborator is all about, then you could effectively mock Apache Geode objects in order to isolate the "*Subject Under Test*" (SUT) and focus on the interaction(s) or outcomes you expect to happen.

With STDG, you don't have to change a bit of configuration to enable mocks in the *Unit Tests* for your Spring Boot applications. You simply only need to annotate the test class with `@EnableGemFireMockObjects`, like so:

Using Mock Apache Geode Objects

```
@RunWith(SpringRunner.class)
@SpringBootTest
class MyApplicationTestClass {

    @Test
    public void someTestCase() {
        // ...
    }

    @Configuration
    @EnableGemFireMockObjects
    static class GeodeConfiguration { }

}
```

Your Spring Boot configuration of Apache Geode will return mock objects for all Apache Geode objects, such as Regions.

Mocking Apache Geode objects even works for objects created from the productivity annotations discussed in the previous sections above.

For example, given the following Spring Boot, Apache Geode `ClientCache` application class:

Main @SpringBootApplication class under test

```
@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
class SpringBootApacheGeodeClientCacheApplication { }
```

The "Customers" Region defined by the `Customer` entity class and created by the `@EnableEntityDefinedRegions` annotation would be a "mock" Region and not an actual Region. You can still inject the Region in your test as before and assert interactions on the Region based on your application workflows:

Using Mock Apache Geode Objects

```
@RunWith(SpringRunner.class)
@SpringBootTest
class MyApplicationTestClass {

    @Resource(name = "Customers")
    private Region<Long, Customer> customers;

    @Test
    public void someTestCase() {

        Customer jonDoe = new Customer(1, "Jon Doe");

        // Use the application in some way and test the interaction on the "Customers"
        Region

        assertThat(this.customers).containsValue(jonDoe);

        // ...
    }
}
```

There are many more things that STDG can do for you in both *Unit & Integration Testing*.

Refer to the [documentation on Unit Testing](#) for more details.

It is possible to [write Integration Tests](#) using STDG as well. Writing *Integration Tests* is an essential concern when you need to assert whether your application OQL Queries are well-formed, for instance. There are many other valid cases where *Integration Testing* is also applicable.

Chapter 7. Externalized Configuration

Like Spring Boot itself (see [here](#)), Spring Boot for Apache Geode (SBDG) supports externalized configuration.

By externalized configuration, we mean configuration metadata stored in a Spring Boot `application.properties` file, for instance. Properties can even be delineated by concern, broken out into individual properties files, that are perhaps only enabled by a specific [Profile](#).

There are many other powerful things you can do, such as, but not limited to, using [placeholders](#) in properties, [encrypting](#) properties, and so on. What we are particularly interested in, in this section, is [type-safety](#).

Like Spring Boot, Spring Boot for Apache Geode provides a hierarchy of classes used to capture configuration for several Apache Geode features in an associated `@ConfigurationProperties` annotated class. Again, the configuration metadata is specified as well-known, documented properties in 1 or more Spring Boot `application.properties` files.

For instance, I may have configured my Spring Boot, `ClientCache` application as follows:

Spring Boot `application.properties` containing Spring Data properties for Apache Geode

```
# Spring Boot application.properties used to configure {geode-name}

spring.data.gemfire.name=MySpringBootApacheGeodeApplication

# Configure general cache properties
spring.data.gemfire.cache.copy-on-read=true
spring.data.gemfire.cache.log-level=debug

# Configure ClientCache specific properties
spring.data.gemfire.cache.client.durable-client-id=123
spring.data.gemfire.cache.client.keep-alive=true

# Configure a log file
spring.data.gemfire.logging.log-file=/path/to/geode.log

# Configure the client's connection Pool to the servers in the cluster
spring.data.gemfire.pool.locators=10.105.120.16[11235],boombox[10334]
```

There are many other properties a user may use to externalize the configuration of their Spring Boot, Apache Geode applications. You may refer to the Spring Data for Apache Geode (SDG) configuration annotations [Javadoc](#) for specific configuration properties as needed. Specifically, review the *"enabling"* annotation attributes.

There may be cases where you require access to the configuration metadata (specified in properties) in your Spring Boot applications themselves, perhaps to further inspect or act on a particular configuration setting.

Of course, you can access any property using Spring's `Environment` abstraction, like so:

Using the Spring `Environment`

```
@Configuration
class GeodeConfiguration {
    void readConfigurationFromEnvironment(Environment environment) {
        boolean copyOnRead = environment.getProperty("spring.data.gemfire.cache.copy-on-read", Boolean.TYPE, false);
    }
}
```

While using the `Environment` is a nice approach, you might need access to additional properties or want to access the property values in a type-safe manner. Therefore, it is now possible, thanks to SBDG's auto-configured configuration processor, to access the configuration metadata using provided `@ConfigurationProperties` classes.

Following on to our example above, I can now do the following:

Using `GemFireProperties`

```
@Component
class MyApplicationComponent {

    @Autowired
    private GemFireProperties gemfireProperties;

    public void someMethodUsingGemFireProperties() {

        boolean copyOnRead = this.gemfireProperties.getCache().isCopyOnRead();

        // do something with `copyOnRead`
    }
}
```

Given a handle to `GemFireProperties`, you can access any of the configuration properties used to configure Apache Geode in a Spring context. You simply only need to autowire an instance of `GemFireProperties` into your application component.

A complete reference to the SBDG provided `@ConfigurationProperties` classes and supporting classes is available [here](#).

7.1. Externalized Configuration of Spring Session

The same capability applies to accessing the externalized configuration of Spring Session when using Apache Geode as your (HTTP) Session state caching provider.

In this case, you simply only need to acquire a reference to an instance of the `SpringSessionProperties` class.

As before, you would specify Spring Session for Apache Geode (SSDG) properties as follows:

*Spring Boot **application.properties** for Spring Session using Apache Geode as the (HTTP) Session state caching provider*

```
# Spring Boot application.properties used to configure {geode-name} as a Session state
caching provider in Spring Session

spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds=300
spring.session.data.gemfire.session.region.name=UserSessions
```

Then, in your application:

*Using **SpringSessionProperties***

```
@Component
class MyApplicationComponent {

    @Autowired
    private SpringSessionProperties springSessionProperties;

    public void someMethodUsingSpringSessionProperties() {

        String sessionRegionName =
this.springSessionProperties.getSession().getRegion().getName();

        // do something with `sessionRegionName`
    }
}
```

Chapter 8. Using Geode Properties

As of Spring Boot for Apache Geode (SBDG) 1.3, it is possible to declare Apache Geode properties from `gemfire.properties` in a Spring Boot `application.properties` file.



A complete list of valid Apache Geode properties can be found in the [User Guide](#).

It should be known that only valid Geode Properties can be declared in `gemfire.properties`, or alternatively, `gfsecurity.properties`.

For example:

Valid `gemfire.properties`

```
# Geode Properties in gemfire.properties

name=ExampleCacheName
log-level=TRACE
enable-time-statistics=true
durable-client-id=123
# ...
```

All of the properties declared in the `gemfire.properties` file shown above correspond to valid Geode Properties. It is illegal to declare properties in a `gemfire.properties` file that are not valid Geode Properties, even if those properties are prefixed with a different qualifier (e.g. "`spring.*`"). Apache Geode is very particular about this and will throw an `IllegalArgumentException` for invalid properties.

For example, given the following `gemfire.properties` file with "`invalid-property`" declared:

Invalid `gemfire.properties`

```
# Geode Properties in gemfire.properties

name=ExampleCacheName
invalid-property=TEST
```

Apache Geode throws an `IllegalArgumentException`:

```
Exception in thread "main" java.lang.IllegalArgumentException: Unknown configuration
attribute name invalid-property.
Valid attribute names are: ack-severe-alert-threshold ack-wait-threshold archive-disk-
space-limit ...
    at o.a.g.internal.AbstractConfig.checkAttributeName(AbstractConfig.java:333)
    at
o.a.g.distributed.internal.AbstractDistributionConfig.checkAttributeName(AbstractDistr
ibutionConfig.java:725)
    at
o.a.g.distributed.internal.AbstractDistributionConfig.getAttributeType(AbstractDistrib
utionConfig.java:887)
    at o.a.g.internal.AbstractConfig.setAttribute(AbstractConfig.java:222)
    at
o.a.g.distributed.internal.DistributionConfigImpl.initialize(DistributionConfigImpl.ja
va:1632)
    at
o.a.g.distributed.internal.DistributionConfigImpl.<init>(DistributionConfigImpl.java:9
94)
    at
o.a.g.distributed.internal.DistributionConfigImpl.<init>(DistributionConfigImpl.java:9
03)
    at
o.a.g.distributed.internal.ConnectionConfigImpl.lambda$new$2(ConnectionConfigImpl.java
:37)
    at
o.a.g.distributed.internal.ConnectionConfigImpl.convert(ConnectionConfigImpl.java:73)
    at
o.a.g.distributed.internal.ConnectionConfigImpl.<init>(ConnectionConfigImpl.java:36)
    at
o.a.g.distributed.internal.InternalDistributedSystem$Builder.build(InternalDistributed
System.java:3004)
    at
o.a.g.distributed.internal.InternalDistributedSystem.connectInternal(InternalDistribut
edSystem.java:269)
    at
o.a.g.cache.client.ClientCacheFactory.connectInternalDistributedSystem(ClientCacheFact
ory.java:280)
    at o.a.g.cache.client.ClientCacheFactory.basicCreate(ClientCacheFactory.java:250)
    at o.a.g.cache.client.ClientCacheFactory.create(ClientCacheFactory.java:216)
    at org.example.app.ApacheGeodeClientCacheApplication.main(...)
```

It is inconvenient to have to separate Apache Geode properties from other application properties, or to have to declare only Apache Geode properties in a `gemfire.properties` file and application properties in a separate properties file, such as Spring Boot `application.properties`.

Additionally, because of Apache Geode's constraint on properties, you are not able to leverage the full power of Spring Boot when composing `application.properties`.

It is well-known that you can include certain properties based on a Spring Profile while excluding other properties. This is essential when properties are environment or context specific.

Of course, users should be aware that Spring Data for Apache Geode (SDG) provide a wide range of properties mapping to Apache Geode properties already.

For example, the SDG `spring.data.gemfire.locators` property maps to the `gemfire.locators` property (or simply, `locators` in `gemfire.properties`) from Apache Geode. Likewise, there are a full set of SDG properties mapping to the corresponding Apache Geode properties in the [Appendix](#).

The Geode Properties shown above can be expressed as SDG Properties in Spring Boot `application.properties` as follows:

Configuring Geode Properties using SDG Properties

```
# Spring Data for {geode-name} properties in application.properties

spring.data.gemfire.name=ExampleCacheName
spring.data.gemfire.cache.log-level=TRACE
spring.data.gemfire.stats.enable-time-statistics=true
spring.data.gemfire.cache.client.durable-client-id=123
# ...
```

However, there are some Apache Geode properties that have no equivalent SDG property, such as `gemfire.groups` (or simply, `groups` in `gemfire.properties`). This is partly due to the fact that many Apache Geode properties are applicable only configured on the server (e.g. `groups` or `enforce-unique-host`).



See the `@EnableGemFireProperties` annotation ([attributes](#)) from SDG for a complete list of Apache Geode properties, which have no corresponding SDG property.

Furthermore, many of the SDG properties also correspond to API calls.

For example, `spring.data.gemfire.cache.client.keep-alive` (see [here](#)) actually translates to the call, `ClientCache.close(boolean keepAlive)` (see [here](#)).

Still, it would be convenient to be able to declare application and Apache Geode properties together, in a single properties file, such as Spring Boot `application.properties`. After all, it is not uncommon to declare JDBC Connection properties in a Spring Boot `application.properties` file.

Therefore, as of SBDG 1.3, it is now possible to declare Apache Geode properties in Spring Boot `application.properties` directly.

For example:


```
# Spring Boot application.properties

server.port=8181
spring.application.name=ExampleApp
gemfire.durable-client-id=123
gemfire.enable-time-statistics=true
```

This is convenient and ideal for several reasons:

1. If you already have a large number of Apache Geode properties declared as `gemfire.` properties, either in `gemfire.properties` or `gfsecurity.properties`, or declared on the Java command-line as JVM System Properties (e.g. `-Dgemfire.name=ExampleCacheName`), then you can reuse these property declarations as is.
2. If you are unfamiliar with SDG's corresponding properties, then you can simply declare Geode Properties instead.
3. You can take advantage of Spring features, such as *Spring Profiles*.
4. You can also use *Property Placeholders* with Geode Properties, e.g. `gemfire.log-level=${external.log-level.property}`



As much as possible, we encourage users to use the SDG provided properties.

However, 1 strict requirement imposed by SBDG is that the Geode Property must have the "`gemfire.`" prefix in a Spring Boot `application.properties` file. This qualifies that the property belongs to Apache Geode. Without, the "`gemfire.`" prefix, the property will not be appropriately applied to the Apache Geode cache instance.

It would be ambiguous if your Spring Boot applications integrated with several technologies, including Apache Geode, and they too had matching properties, e.g. `bind-address` or `log-file`, perhaps.

SBDG makes a best attempt to log warnings when the Geode Property is invalid or not set. For example, the following Geode Property would result in a log warning:

Invalid Apache Geode Property

```
# Spring Boot application.properties

spring.application.name=ExampleApp
gemfire.non-existing-property=TEST
```

The resulting warning appearing in the log would read:

```
[gemfire.non-existing-property] is not a valid Apache Geode property
```

If a Geode Property is not properly set, then the following warning will be logged:

Apache Geode Property [gemfire.security-manager] was not set

With regards to the 3rd point, you can now compose and declare Geode Properties based on context (e.g. your application environment) with Spring Profiles.

For example, you might start with a base set of properties in Spring Boot `application.properties`:

Base Properties

```
server.port=8181
spring.application.name=ExampleApp
gemfire.durable-client-id=123
gemfire.enable-time-statistics=false
```

And then begin to vary the properties by environment:

QA Properties

```
# Spring Boot application-qa.properties

server.port=9191
spring.application.name=TestApp
gemfire.enable-time-statistics=true
gemfire.enable-network-partition-detection=true
gemfire.groups=QA
# ...
```

Or in production:

PROD Properties

```
# Spring Boot application-prod.properties

server.port=80
spring.application.name=ProductionApp
gemfire.archive-disk-space-limit=1000
gemfire.archive-file-size-limit=50
gemfire.enforce-unique-host=true
gemfire.groups=PROD
# ...
```

It is then a simple matter to apply the appropriate set of properties by configuring the Spring Profile by using: `-Dspring.profiles.active=prod`. It is also possible to enable more than 1 profile at a time by using: `-Dspring.profiles.active=profile1,profile2,...,profileN`

If both `spring.data.gemfire.*` properties and the matching Apache Geode properties are declared in Spring Boot `application.properties`, then the SDG properties take precedence.

If a property is specified more than once, as would potentially be the case when composing multiple `application.properties` files and you enable more than 1 Spring Profile at time, then the last property declaration wins. In the example shown above, the value for `gemfire.groups` would be `PROD` when `-Dspring.profiles.active=qa,prod` is configured.

For example, given the following Spring Boot `application.properties`:

Property Precedence

```
# Spring Boot application.properties

gemfire.durable-client-id=123
spring.data.gemfire.cache.client.durable-client-id=987
```

Then the `durable-client-id` will be `987`. It does not matter which order the SDG or Apache Geode properties are declared in `application.properties`, the matching SDG property will override the Apache Geode property when duplicates are found.

Finally, it is not possible to refer to Geode Properties declared in Spring Boot `application.properties` with the SBDG `GemFireProperties` class (See [Javadoc](#)).

For example, given:

Geode Properties declared in Spring Boot `application.properties`

```
# Spring Boot application.properties

gemfire.name=TestCacheName
```

The following assertion holds:

```
import org.springframework.geode.boot.autoconfigure.configuration.GemFireProperties;

@RunWith(SpringRunner.class)
@SpringBootTest
class GemFirePropertiesTestSuite {

    @Autowired
    private GemFireProperties gemfireProperties;

    @Test
    public void gemfirePropertiesTestCase() {

        assertThat(this.gemfireProperties.getCache().getName()).isNotEqualTo("TestCacheName");
    }
}
```



`application.properties` can be declared in the `@SpringBootTest` annotation. For example, `gemfire.name` could have been declared in the annotation using the declaration, `@SpringBootTest(properties = { "gemfire.name=TestCacheName" })`, for testing purposes instead of declaring the property in a separate `application.properties` file.

Only `spring.data.gemfire.*` prefixed properties are mapped to the SBDG `GemFireProperties` class hierarchy.



Again, prefer SDG Properties over Geode Properties. See SDG properties reference in the [Appendix](#).

Chapter 9. Caching with Apache Geode

One of the easiest, quickest and least invasive ways to get started using Apache Geode in your Spring Boot applications is to use Apache Geode as a [caching provider](#) in [Spring's Cache Abstraction](#). SDG [enables](#) Apache Geode to function as a *caching provider* in Spring's Cache Abstraction.



See the *Spring Data for Apache Geode Reference Guide* for more details on the [support](#) and [configuration](#) of Apache Geode as a *caching provider* in Spring's Cache Abstraction.



Make sure you thoroughly understand the [concepts](#) behind Spring's Cache Abstraction before you continue.



You can also refer to the relevant section on [Caching](#) in *Spring Boot's Reference Documentation*. *Spring Boot* even provides *auto-configuration* support for a few, simple [caching providers](#) out-of-the-box.

Indeed, *caching* can be a very effective *software design pattern* to avoid the cost of invoking a potentially expensive operation when, given the same input, the operation yields the same output every time.

Some classic examples of caching include, but are not limited to: looking up a customer by name or account number, looking up a book by ISBN, geocoding a physical address, caching the calculation of a person's credit score when the person applies for a financial loan.

If you need the proven power of an enterprise-class caching solution, with strong consistency, high availability, low latency and multi-site (WAN) capabilities, then you should consider [Apache Geode](#), or alternatively, VMWare, Inc. offers a commercial solution built on Apache Geode called, VMware Tanzu GemFire.

Spring's [declarative, annotation-based caching](#) makes it extremely simple to get started with caching, which is as easy as annotating your application components with the appropriate Spring cache annotations.



Spring's declarative, annotation-based caching also [supports](#) JCache (JSR-107) annotations.

For example, suppose you want to cache the results of determining a person's eligibility when applying for a financial loan. A person's financial status is unlikely to change in the time that the computer runs the algorithms to compute a person's eligibility after all the financial information for the person has been collected and submitted for review and processing.

Our application might consist of a financial loan service to process a person's eligibility over a given period of time:

```
@Service
class FinancialLoanApplicationService {

    @Cacheable("EligibilityDecisions")
    EligibilityDecision processEligibility(Person person, Timespan timespan) {
        // ...
    }
}
```

Notice the `@Cacheable` annotation declared on the `processEligibility(:Person, :Timespan)` method of our service class.

When the `FinancialLoanApplicationService.processEligibility(..)` method is called, Spring's caching infrastructure first consults the `"EligibilityDecisions"` cache to determine if a decision has already been computed for the given person within the given span of time. If the person's eligibility in the given time frame has already been determined, then the existing decision is returned from the cache. Otherwise, the `processEligibility(..)` method will be invoked and the result of the method will be cached when the method returns, before returning the decision to the caller.

Spring Boot for Apache Geode *auto-configures* Apache Geode as the *caching provider* when Apache Geode is declared on the application classpath, and when no other *caching provider* (e.g. Redis) has been configured.

If Spring Boot for Apache Geode detects that another *cache provider* has already been configured, then Apache Geode will not function as the *caching provider* for the application. This allows users to configure another store, e.g. Redis, as the *caching provider* and perhaps use Apache Geode as your application's persistent store.

The only other requirement to enable caching in a Spring Boot application is for the declared caches (as specified in Spring's or JSR-107's caching annotations) to have been created and already exist, especially before the operation on which caching has been applied is invoked. This means the backend data store must provide the data structure serving as the `"cache"`. For Apache Geode this means a cache `Region`.

To configure the necessary Regions backing the caches declared in Spring's cache annotations, this is as simple as using Spring Data for Apache Geode's `@EnableCachingDefinedRegions` annotation.

The complete Spring Boot application looks like this:

```
package example.app;

@SpringBootApplication
@EnableCachingDefinedRegions
class FinancialLoanApplication {

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



The `FinancialLoanApplicationService` is picked up by Spring's classpath component scan since this class is annotated with Spring's `@Service` stereotype annotation.



You can set the `DataPolicy` of the Region created through the `@EnableCachingDefinedRegions` annotation by setting the `clientRegionShortcut` to a valid enumerated value.



Spring Boot for Apache Geode does not recognize nor apply the `spring.cache.cache-names` property. Instead, you should use SDG's `@EnableCachingDefinedRegions` on an appropriate Spring Boot application `@Configuration` class.

9.1. Look-Aside Caching, Near Caching, Inline Caching and Multi-Site Caching

Four different types of caching patterns can be applied with Spring when using Apache Geode for your application caching needs.

The 4 primary caching patterns include:

- *Look-Aside Caching*
- *Near Caching*
- *Inline Caching*
- *Multi-Site Caching*

Typically, when most users think of caching, they are thinking of *Look-Aside Caching*. This is the default caching pattern applied by *Spring's Cache Abstraction*.

In a nutshell, *Near Caching* keeps the data closer to where the data is used thereby improving on performance due to lower latencies when data is needed (i.e. no extra network hops). This also improves application throughput, i.e. the amount of work completed in a given period of time.

Within *Inline Caching*, developers have a choice between synchronous (*Read/Write-Through*) and asynchronous (*Write-Behind*) configurations depending on the application use case and

requirements. Synchronous, Read/Write-Through *Inline Caching* is necessary if consistency is a concern. Asynchronous, Write-Behind *Inline Caching* is applicable if throughput and low-latency are a priority.

Within *Multi-Site Caching*, there are *Active-Passive* and *Active-Active* arrangements. More details on *Multi-Site Caching* will be presented in a later release.

9.1.1. Look-Aside Caching



Refer to the corresponding Sample [Guide](#) and [Code](#) to see *Look-Aside Caching* using Apache Geode in action!

The caching pattern demonstrated in the example above is a form of *Look-Aside Caching*.

Essentially, the data of interest is searched for in the cache first, before calling a potentially expensive operation, e.g. like an operation that makes an IO or network bound request resulting in either a blocking, or a latency sensitive computation.

If the data can be found in the cache (stored in-memory to reduce latency) then the data is returned without ever invoking the expensive operation. If the data cannot be found in the cache, then the operation must be invoked. However, before returning, the result of the operation is cached for subsequent requests when the the same input is requested again, by another caller resulting in much improved response times.

Again, typical *Look-Aside Caching* pattern applied in your application code looks similar to the following:

Look-Aside Caching Pattern Applied

```
@Service
class CustomerService {

    private final CustomerRepository customerRepository;

    @Cacheable("Customers")
    Customer findByAccount(Account account) {

        // pre-processing logic here

        Customer customer = customerRepository.findByAccountNumber(account.getNumber());

        // post-processing logic here

        return customer;
    }
}
```

In this design, the `CustomerRepository` is perhaps a JDBC or JPA/Hibernate backed implementation accessing the external data source (i.e. RDBMS) directly. The `@Cacheable` annotation wraps, or

"decorates", the `findByAccount(:Account):Customer` operation to provide caching facilities.



This operation may be expensive because it might validate the Customer's Account before looking up the Customer, pull multiple bits of information to retrieve the Customer record, and so on, hence the need for caching.

9.1.2. Near Caching



Refer to the corresponding Sample [Guide](#) and [Code](#) to see *Near Caching* using Apache Geode in action!

Near Caching is another pattern of caching where the cache is collocated with the application. This is useful when the caching technology is configured using a client/server arrangement.

We already mentioned that Spring Boot for Apache Geode [provides](#) an *auto-configured*, `ClientCache` instance, out-of-the-box, by default. A `ClientCache` instance is most effective when the data access operations, including cache access, is distributed to the servers in a cluster accessible by the client, and in most cases, multiple clients. This allows other cache client applications to access the same data. However, this also means the application will incur a network hop penalty to evaluate the presence of the data in the cache.

To help avoid the cost of this network hop in a client/server topology, a local cache can be established, which maintains a subset of the data in the corresponding server-side cache (i.e. Region). Therefore, the client cache only contains the data of interests to the application. This "local" cache (i.e. client-side Region) is consulted before forwarding the lookup request to the server.

To enable *Near Caching* when using either Apache Geode, simply change the Region's (i.e. the `Cache` in Spring's Cache Abstraction) data management policy from `PROXY` (the default) to `CACHING_PROXY`, like so:

Enabling Near Caching using Apache Geode

```
@SpringBootApplication
@EnableCachingDefinedRegions(clientRegionShortcut =
ClientRegionShortcut.CACHING_PROXY)
class FinancialLoanApplication {

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



The default, client Region data management policy is `ClientRegionShortcut.PROXY`. As such, all data access operations are immediately forwarded to the server.



Also see the Apache Geode documentation concerning [Client/Server Event Distribution](#) and specifically, "*Client Interest Registration on the Server*" when client CACHING_PROXY Regions to manage state in addition to the corresponding server-side Region. This is necessary to receive updates on entries in the Region that might have been changed by other clients accessing the same data.

9.1.3. Inline Caching

The next pattern of caching we will discuss in this chapter is *Inline Caching*.

There are two different configurations of *Inline Caching* that developers can apply to their Spring Boot applications when using the *Inline Caching* pattern: Synchronous (*Read/Write-Through*) and Asynchronous (*Write-Behind*).



Asynchronous (currently) only offers write capabilities, from the cache to the backend, external data source. There is no option to asynchronously and automatically load the cache when the value becomes available in the backend, external data source.

Synchronous Inline Caching



Refer to the corresponding Sample [Guide](#) and [Code](#) to see *Inline Caching* using Apache Geode in action!

When employing *Inline Caching* and a cache miss occurs, the application service method may still not be invoked since a cache can be configured to invoke a loader to load the missing entry from an backend, external data source.

With Apache Geode the cache, or using Apache Geode terminology, the Region, can be configured with a [CacheLoader](#). A [CacheLoader](#) is implemented to retrieve missing values from an external data source, which could be an RDBMS or any other type of data store (e.g. another NoSQL data store like Apache Cassandra, MongoDB or Neo4j), when a cache miss occurs.



See the Apache Geode User Guide on [Data Loaders](#) for more details.

Likewise, an Apache Geode Region can also be configured with a [CacheWriter](#). A [CacheWriter](#) is responsible for writing an entry put into the Region to the backend data store, such as an RDBMS. This is referred to as a "*write-through*" operation because it is synchronous. If the backend data store fails to be updated then the entry will not be stored in the Region. This helps to ensure consistency between the backend data store and the Apache Geode Region.



It is also possible to implement *Inline-Caching* using *asynchronous*, *write-behind* operations by registering an [AsyncEventListener](#) on an [AEQ](#) attached to a server-side Region. You should consult the Apache Geode User Guide for more [details](#). We cover *asynchronous*, *write-behind* *Inline Caching* in the next section.

The typical pattern of *Inline Caching* when applied to application code looks similar to the

following:

Inline Caching Pattern Applied

```
@Service
class CustomerService {

    private CustomerRepository customerRepository;

    Customer findByAccount(Account account) {

        // pre-processing logic here

        Customer customer = customerRepository.findByAccountNumber(account.getNumber());

        // post-processing logic here.

        return customer;
    }
}
```

The main difference is, there are no Spring or JSR-107 caching annotations applied to the application's service methods and the `CustomerRepository` is accessing Apache Geode directly and NOT the RDBMS.

Implementing CacheLoaders & CacheWriters for Inline Caching

You can use Spring to configure a `CacheLoader` or `CacheWriter` as a bean in the Spring `ApplicationContext` and then wire the loader and/or writer to a Region. Given the `CacheLoader` or `CacheWriter` is a Spring bean like any other bean in the Spring `ApplicationContext`, you can inject any `DataSource` you like into the Loader/Writer.

While you can configure client Regions with `CacheLoaders` and `CacheWriters`, it is typically more common to configure the corresponding server-side Region; for example:

```

@SpringBootApplication
@CacheServerApplication
class FinancialLoanApplicationServer {

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

    @Bean("EligibilityDecisions")
    PartitionedRegionFactoryBean<Object, Object> eligibilityDecisionsRegion(
        GemFireCache gemfireCache, CacheLoader decisionManagementSystemLoader,
        CacheWriter decisionManagementSystemWriter) {

        PartitionedRegionFactoryBean<?, EligibilityDecision>
eligibilityDecisionsRegion =
        new PartitionedRegionFactoryBean<>();

        eligibilityDecisionsRegion.setCache(gemfireCache);
        eligibilityDecisionsRegion.setCacheLoader(decisionManagementSystemLoader);
        eligibilityDecisionsRegion.setCacheWriter(decisionManagementSystemWriter);
        eligibilityDecisionsRegion.setPersistent(false);

        return eligibilityDecisionsRegion;
    }

    @Bean
    CacheLoader<?, EligibilityDecision> decisionManagementSystemLoader(
        DataSource dataSource) {

        return new DecisionManagementSystemLoader(dataSource);
    }

    @Bean
    CacheWriter<?, EligibilityDecision> decisionManagementSystemWriter(
        DataSource dataSource) {

        return new DecisionManagementSystemWriter(dataSource);
    }

    @Bean
    DataSource dataSource() {
        // ...
    }
}

```

Then, you would implement the `CacheLoader` and `CacheWriter` interfaces as appropriate:

```
class DecisionManagementSystemLoader implements CacheLoader<?, EligibilityDecision> {  
  
    private final DataSource dataSource;  
  
    DecisionManagementSystemLoader(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    public EligibilityDecision load(LoadHelper<?, EligibilityDecision> helper) {  
  
        Object key = helper.getKey();  
  
        // Use the configured DataSource to load the value identified by the key from a  
        backend, external data store.  
    }  
}
```



SBDG provides the `org.springframework.geode.cache.support.CacheLoaderSupport` `@FunctionalInterface` to conveniently implement application `CacheLoaders`.

If the configured `CacheLoader` still cannot resolve the value, then the cache lookup operation results in a miss and the application service method will then be invoked to compute the value.

```
class DecisionManagementSystemWriter implements CacheWriter<?, EligibilityDecision> {  
  
    private final DataSource dataSource;  
  
    DecisionManagementSystemWriter(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    public void beforeCreate(EntryEvent<?, EligibilityDecision> entryEvent) {  
        // Use configured DataSource to save (e.g. INSERT) the entry value into the  
        backend data store  
    }  
  
    public void beforeUpdate(EntryEvent<?, EligibilityDecision> entryEvent) {  
        // Use the configured DataSource to save (e.g. UPDATE or UPSERT) the entry value  
        into the backend data store  
    }  
  
    public void beforeDestroy(EntryEvent<?, EligibilityDecision> entryEvent) {  
        // Use the configured DataSource to delete (i.e. DELETE) the entry value from the  
        backend data store  
    }  
  
    // ...  
}
```



SBDG provides the `org.springframework.geode.cache.support.CacheWriterSupport` interface to conveniently implement application `CacheWriters`.



Of course, your `CacheWriter` implementation can use any data access technology to interface with your backend data store (e.g. JDBC, Spring's `JdbcTemplate`, JPA/Hibernate, etc). It is not limited to only using a `javax.sql.DataSource`. In fact, we will present another, more useful and convenient approach to implementing *Inline Caching* in the next section.

Inline Caching using Spring Data Repositories

Spring Boot for Apache Geode (SBDG) offers dedicated support to configure *Inline Caching* using Spring Data Repositories.

This is very powerful because it allows you to:

1. Access any backend data store supported by Spring Data (e.g. Redis for Key/Value or other data structures, MongoDB for Documents, Neo4j for Graphs, Elasticsearch for Search, and so on).
2. Use complex mapping strategies (e.g. ORM provided by JPA/Hibernate).

It is our belief that users should be storing data where it is most easily accessible. If you are

accessing and processing Documents, then MongoDB, Couchbase or another document store is probably going to be the most logical choice to manage your application's Documents.

However, this does not mean you have to give up Apache Geode in your application/system architecture. You can leverage each data store for what it is good at. While MongoDB is excellent at handling documents, Apache Geode is a highly valuable choice for consistency, high availability, low-latency/high-throughput, multi-site, scale-out application use cases.

As such, using Apache Geode's `CacheLoader/CacheWriter` functionality provides a nice integration point between itself and other data stores to best serve your application's use case and requirements.

EXAMPLE

Let's say you are using JPA/Hibernate to access (store and read) data managed in an Oracle Database. Then, you can configure Apache Geode to read/write-through to the backend Oracle Database when performing cache (Region) operations by delegating to a Spring Data JPA Repository.

The configuration might look something like:

Inline Caching configuration using SBDG

```
@SpringBootApplication
@EntityScan(basePackageClasses = Customer.class)
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableJpaRepositories(basePackageClasses = CustomerRepository.class)
class SpringBootOracleDatabaseApacheGeodeApplication {

    @Bean
    InlineCachingRegionConfigurer<Customer, Long>
    inlineCachingForCustomersRegionConfigurer(
        CustomerRepository customerRepository) {

        return new InlineCachingRegionConfigurer<>(customerRepository,
        Predicate.isEqual("Customers"));
    }
}
```

Out-of-the-box, SBDG provides the `InlineCachingRegionConfigurer<ENTITY, ID>` interface.

Given a `Predicate` to express the criteria used to match the target Region by name and a Spring Data `CrudRepository`, the `InlineCachingRegionConfigurer` will configure and adapt the Spring Data `CrudRepository` as a `CacheLoader` and `CacheWriter` registered on the Region (e.g. "Customers") to enable *Inline Caching* functionality.

You simply only need to declare `InlineCachingRegionConfigurer` as a bean in the Spring `ApplicationContext` and make the association between the Region (by name) and the appropriate Spring Data `CrudRepository`.

In this example, we used JPA and Spring Data JPA to store/retrieve the data in the cache (Region)

to/from a backend database. But, you can inject any Spring Data Repository for any data store (e.g. Redis, MongoDB, etc) that supports the Spring Data Repository abstraction.



If you only want to support one way data access operations when using *Inline Caching*, then you can use either the `RepositoryCacheLoaderRegionConfigurer` for reads or the `RepositoryCacheWriterRegionConfigurer` for writes, instead of the `InlineCachingRegionConfigurer`, which supports both reads and writes.



To see a similar implementation of *Inline Caching* using a Database (In-Memory, HSQLDB Database) in action, have a look at this [test class](#) from the SBDG test suite. A dedicated sample will be provided in a future release.

Asynchronous Inline Caching



Refer to the corresponding Sample [Guide](#) and [Code](#) to see *Asynchronous Inline Caching* using Apache Geode in action!

If consistency between the cache and your external, backend data source is not a concern, and you only need to write from the cache to the backend data store periodically, then you can employ asynchronous (*Write-Behind*) *Inline Caching*.

As the term "*Write-Behind*" implies, a write to the backend data store is asynchronous and not strictly tied to the cache operation. As a result, the backend data store will be in an "*eventually consistent*" state since the cache is primarily used by the application at runtime to access and manage data. In this case, the backend data store is used to persist the state of the cache, and that of the application, at periodic intervals.

Of course, if multiple applications are updating the backend data store concurrently, you could combine a `CacheLoader` to synchronously "*Read-Through*" to the backend data store and keep the cache up-to-date as well as asynchronously *Write-Behind* from the cache to the backend data store when the cache is updated to eventually inform other interested applications of data changes. In this capacity, the backend data store is still the primary *System of Record* (SOR).

If data processing is not time sensitive, you can gain a performance advantage from periodic, quantity and/or time-based batch updates.

Implementing an AsyncEventListener for Inline Caching

If you were to configure asynchronous (*Write-Behind*) *Inline Caching* by hand, then you would need to do all of the following yourself:

1. Implement an `AsyncEventListener` to write to an external, backend data source on cache events
2. Configure, create and register the listener with an `AsyncEventQueue` (AEQ)
3. Create a Region serving as the source of cache events and attach the AEQ

The advantage of this approach is you have access to and control over low-level configuration details. The disadvantage, of course, is with more moving parts, it is easier to mess things up.

Following on from our synchronous (*Read/Write-Through*) *Inline Caching* examples from the prior sections above, our `AsyncEventListener` implementation might appear as follows:

Example `AsyncEventListener` for *Async Inline Caching*

```
@Component
class ExampleAsyncEventListener implements AsyncEventListener {

    private final DataSource dataSource;

    ExampleAsyncEventListener(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    @Override
    public boolean processEvents(List<AsyncEvent> events) {

        // Iterate over the ordered AsyncEvents and use the DataSource
        // to write to the external, backend DataSource

    }
}
```



Instead of injecting a `DataSource` into your `AsyncEventListener` directly, you could use JDBC, Spring's `JdbcTemplate`, JPA/Hibernate or another data access API/Framework. Further below, we will show how SBDG simplifies the `AsyncEventListener` implementation by using Spring Data *Repositories*.

Then, we need to register this listener with a `AsyncEventQueue` (AEQ) (#2) and attach it to the target Region that will be the source of the cache events we want to persist asynchronously (#3):

```
@Configuration
@PeerCacheApplication
class GeodeConfiguration {

    @Bean
    DataSource exampleDataSource() {
        // Configure and construct a data store specific DataSource
    }

    @Bean
    ExampleAsyncEventListener exampleAsyncEventListener(DataSource dataSource) {
        return new ExampleAsyncEventListener(dataSource);
    }

    @Bean
    AsyncEventQueueFactoryBean exampleAsyncEventQueue(Cache peerCache,
ExampleAsyncEventListener listener) {

        AsyncEventQueueFactoryBean asyncEventQueue = new
AsyncEventQueueFactoryBean(peerCache, listener);

        asyncEventQueue.setBatchConflationEnabled(true);
        asyncEventQueue.setBatchSize(50);
        asyncEventQueue.setBatchTimeInterval(15000); // 15 seconds
        asyncEventQueue.setMaximumQueueMemory(64); // 64 MB
        // ...

        return asyncEventQueue;
    }

    @Bean("Example")
    PartitionedRegionFactoryBean<?, ?> exampleRegion(Cache peerCache, AsyncEventQueue
queue) {

        PartitionedRegionFactoryBean<?, ?> exampleRegion = new
PartitionedRegionFactoryBean<>();

        exampleRegion.setAsyncEventQueues(ArrayUtils.asArray(queue));
        exampleRegion.setCache(peerCache);
        // ...

        return exampleRegion;
    }
}
```

While this approach affords you the developer a lot of control over the (low-level) configuration, in addition to your `AsyncEventListener` implementation, this is a lot of boilerplate code.



See the [Javadoc](#) on SDG's [AsyncEventQueueFactoryBean](#) for more details on the configuration of the AEQ.



See Apache Geode's [User Guide](#) for more details on AEQs and listeners.

Fortunately, with SBDG, there is a better way!

Asynchronous Inline Caching using Spring Data Repositories

The implementation and configuration of the [AsyncEventListener](#) as well as the AEQ shown above can be simplified as follows:

Using SBDG to configure Asynchronous (Write-Behind) Inline Caching

```
@SpringBootApplication
@EntityScan(basePackageClasses = ExampleEntity.class)
@EnableJpaRepositories(basePackageClasses = ExampleRepository.class)
@EnableEntityDefinedRegions(basePackageClasses = ExampleEntity.class)
class ExampleSpringBootApacheGeodeAsyncInlineCachingApplication {

    @Bean
    AsyncInlineCachingRegionConfigurer asyncInlineCachingRegionConfigurer(
        CrudRepository<ExampleEntity, Long> repository) {

        return AsyncInlineCachingRegionConfigurer.create(repository, "Example")
            .withQueueBatchConflationEnabled()
            .withQueueBatchSize(50)
            .withQueueBatchTimeInterval(Duration.ofSeconds(15))
            .withQueueMaxMemory(64);
    }
}
```

The [AsyncInlineCachingRegionConfigurer.create\(..\)](#) method is overloaded to accept a [Predicate](#) in place of the [String](#) in order to express more powerful matching logic, programmatically, identifying the target Region (by name) on which to configure asynchronous *Inline Caching* functionality.

The [AsyncInlineCachingRegionConfigurer](#) uses the [Builder Software Design Pattern](#) and [withQueue*\(..\)](#) builder methods to configure the underlying [AsyncEventQueue](#) (AEQ) when the queue's configuration deviates from the defaults, as specified by Apache Geode.

Under-the-hood, the [AsyncInlineCachingRegionConfigurer](#) constructs a new instance of the [RepositoryAsyncEventListener](#) class initialized with the given Spring Data [CrudRepository](#). The [RegionConfigurer](#) then registers the listener with the AEQ and attaches it to the target [Region](#).

With the power of Spring Boot *auto-configuration* and SBDG, the configuration is much more concise and intuitive.

About [RepositoryAsyncEventListener](#)

The SBDG [RepositoryAsyncEventListener](#) class is the magic sauce behind the integration of the cache

with an external, backend data source.

The listener is a specialized `Adapter` that processes `AsyncEvents` by invoking an appropriate `CrudRepository` method based on the cache operation. The listener requires an instance of `CrudRepository`. As such, the listener supports any external, backend data source supported by Spring Data's *Repository* abstraction.

Of course, backend data store, data access operations (e.g. INSERT, UPDATE, DELETE, etc) triggered by cache events are performed asynchronously from the cache operation. This means the state of the cache and backend data store will be "*eventually consistent*".

ERROR HANDLING

Given the complex nature of "*eventually consistent*" systems and asynchronous concurrent processing, the `RepositoryAsyncEventListener` allows users to register a custom `AsyncEventHandler` to handle the errors that occur during processing of `AsyncEvents`, perhaps due to a faulty backend data store data access operation (e.g. `OptimisticLockingFailureException`), in an application relevant way.

The `AsyncEventHandler` interface is a `java.util.function.Function` implementation and `@FunctionalInterface` defined as:

AsyncEventHandler interface definition

```
@FunctionalInterface
interface AsyncEventHandler implements Function<AsyncEventError, Boolean> { }
```

The `AsyncEventError` class encapsulates `AsyncEvent` along with the `Throwable` that was thrown while processing the event.

Since the `AsyncEventHandler` interface implements `Function`, then you would override the `apply(:AsyncEventError)` method to handle the error with application-specific actions. The handler returns a `Boolean` to indicate whether it was able to handle the error or not.

Custom `AsyncEventHandler` implementation

```
class CustomAsyncEventHandler implements AsyncEventHandler {

    @Override
    public Boolean apply(AsyncEventError error) {

        if (error.getCause() instanceof OptimisticLockingFailureException) {
            // handle optimistic locking failure if you can
            return true; // if error was successfully handled.
        }
        else if (error.getCause() instanceof IncorrectResultSizeDataAccessException) {
            // handle no row or too many row update if you can
            return true; // if error was successfully handled.
        }

        return false;
    }
}
```

It is easy to configure the `RepositoryAsyncEventListener` with your custom `AsyncEventHandler` using the `AsyncInlineCachingRegionConfigurer`, like so:

Configuring a custom `AsyncEventHandler`

```
@Configuration
class GeodeConfiguration {

    @Bean
    CustomAsyncEventHandler customAsyncEventHandler() {
        return new CustomAsyncEventHandler();
    }

    @Bean
    AsyncInlineCachingRegionConfigurer asyncInlineCachingRegionConfigurer(
        CrudRepository<?, ?> repository,
        CustomAsyncEventHandler errorHandler
    ) {

        return AsyncInlineCachingRegionConfigurer.create(repository, "Example")
            .withAsyncEventHandler(errorHandler);
    }
}
```

Also, since `AsyncEventHandler` implements `Function`, you can "*compose*" multiple error handlers using `Function.andThen(:Function)`.

SUPPORTED CACHE OPERATIONS

By default, the `RepositoryAsyncEventListener` handles `CREATE`, `UPDATE` and `REMOVE` cache event, entry

operations.

CREATE and UPDATE translates to `CrudRepository.save(entity)` where the `entity` is derived from `AsyncEvent.getDeserializedValue()`.

REMOVE translates to `CrudRepository.delete(entity)` where the `entity` is derived from `AsyncEvent.getDeserializedValue()`.

The cache `Operation` to `CrudRepository` method is supported by the `AsyncEventOperationRepositoryFunction` interface, which implements `java.util.function.Function` and is a `@FunctionalInterface`.

This interface becomes useful if and when you want to implement `CrudRepository` method invocations for other `AsyncEvent Operations` not handled by SBDG's `RepositoryAsyncEventListener` out-of-the-box.

The `AsyncEventOperationRepositoryFunction` interface is defined as:

AsyncEventOperationRepositoryFunction interface definition

```
@FunctionalInterface
interface AsyncEventOperationRepositoryFunction<T, ID> implements
Function<AsyncEvent<ID, T>, Boolean> {

    default boolean canProcess(AsyncEvent<ID, T> event) {
        return false;
    }
}
```

`T` is the class type of the entity and `ID` is the class type of the entity's identifier (ID), possibly declared with Spring Data's `org.springframework.data.annotation.Id` annotation.

For convenience, SBDG provides the `AbstractAsyncEventOperationRepositoryFunction` class for extension, where you would provide implementations for the `cacheProcess(:AsyncEvent)` and `doRepositoryOp(entity)` methods.



The `AsyncEventOperationRepositoryFunction.apply(:AsyncEvent)` method is already implemented in terms of `canProcess(:AsyncEvent)`, `resolveEntity(:AsyncEvent)`, `doRepositoryOp(entity)`, and catching and handling any `Throwable` (errors) by calling the configured `AsyncEventHandler`.

For example, you might want to handle `Operation.INVALIDATE` cache events as well, deleting the entity from the backend data store by invoking the `CrudRepository.delete(entity)` method:

```
@Component
class InvalidateAsyncEventRepositoryFunction
    extends
RepositoryAsyncEventListener.AbstractAsyncEventOperationRepositoryFunction<?, ?> {

    InvalidateAsyncEventRepositoryFunction(RepositoryAsyncEventListener<?, ?>
listener) {
        super(listener);
    }

    @Override
    public boolean canProcess(AsyncEvent<?, ?> event) {
        return event != null && Operation.INVALIDATE.equals(event.getOperation());
    }

    @Override
    protected Object doRepositoryOperation(Object entity) {
        getRepository().delete(entity);
        return null;
    }
}
```

You can then register your user-defined, `AsyncEventOperationRepositoryFunction` (i.e. `InvalidateAsyncEventRepositoryFunction`) with the `RepositoryAsyncEventListener` by using the `AsyncInlineCachingRegionConfigurer`, like so:

```
import org.springframework.geode.cache.RepositoryAsyncEventListener;@Configuration
class GeodeConfiguration {

    @Bean
    AsyncInlineCachingRegionConfigurer asyncInlineCachingRegionConfigurer(
        CrudRepository<?, ?> repository,
        CustomerAsyncEventErrorHandler errorHandler
    ) {

        return AsyncInlineCachingRegionConfigurer.create(repository, "ExampleRegion")
            .applyToListener(listener -> {

                if (listener instanceof RepositoryAsyncEventListener) {

                    RepositoryAsyncEventListener<?, ?> repositoryListener =
                        (RepositoryAsyncEventListener<?, ?>) listener;

                    repositoryListener.register(new
InvalidAsyncEventRepositoryFunction(repositoryListener));
                }

                return listener;
            });
    }
}
```

This same technique can be applied to `CREATE`, `UPDATE` and `REMOVE` cache operations as well, effectively overriding the default behavior for this cache operations handled by SBDG out-of-the-box.

About `AsyncInlineCachingRegionConfigurer`

As we saw in the previous section, it is possible to intercept and post-process key components constructed and configured by the `AsyncInlineCachingRegionConfigurer` class during initialization.

Out-of-the-box, SBDG's allows you to intercept and post-process the `AsyncEventListener` (e.g. `RepositoryAsyncEventListener`), `AsyncEventQueueFactory` and even the `AsyncEventQueue`, created by the `AsyncInlineCachingRegionConfigurer` (a `SDG RegionConfigurer`) during Spring `ApplicationContext`, bean initialization.

The `AsyncInlineCachingRegionConfigurer` class provides the builder methods listed below to intercept and post-process any of the following Apache Geode objects:

- `applyToListener(:Function<AsyncEventListener, AsyncEventListener>)`
- `applyToQueue(:Function<AsyncEventQueue, AsyncEventQueue>)`
- `applyToQueueFactory(:Function<AsyncEventQueueFactory, AsyncEventQueueFactory>)`

All of these "apply" methods accept a `java.util.function.Function` that "applies" the logic of the

Function to the Apache Geode object (e.g. [AsyncEventListener](#)), returning the object as a result.



The Apache Geode object returned by the **Function** may be the same object, a proxy, or a completely new object. Essentially, the returned object can be anything you want. This is the fundamental premise behind *Aspect-Oriented Programming* (AOP) and the [Decorator Software Design Pattern](#).

These "apply" methods and the supplied **Function** allow you to decorate, enhance, post-process, whatever you want to, to the Apache Geode objects created by the listener.

Of course, the [AsyncInlineCachingRegionConfigurer](#) strictly adheres to the [Open/Close Principle](#) as well, and is therefore flexibly extensible.

9.1.4. Multi-Site Caching

The final pattern of caching presented in this chapter is *Multi-Site Caching*.

As described above, there are 2 configuration arrangements depending on your application usage patterns, requirements and user demographic: *Active-Active* & *Active-Passive*.

Multi-Site Caching along with *Active-Active* and *Active-Passive* configuration arrangements will be described in more detail in the Sample [Guide](#). Also, be sure to review the Sample [Code](#).

9.2. Advanced Caching Configuration

Apache Geode supports additional caching capabilities to manage the entries stored in the cache.

As you can imagine, given that cache entries are stored in-memory, it becomes important to monitor and manage the available memory wisely. After all, by default, Apache Geode stores data in the JVM Heap.

Several techniques can be employed to more effectively manage memory, such as using [Eviction](#), possibly [overflowing data to disk](#), configuring both entry *Idle-Timeout* (TTI) as well as *Time-To-Live* (TTL) [Expiration policies](#), configuring [Compression](#), and using [Off-Heap](#), or main memory.

There are several other strategies that can be used as well, as described in [Managing Heap and Off-heap Memory](#).

While this is well beyond the scope of this document, know that Spring Data for Apache Geode makes all of these [configuration options](#) available and simple to use.

9.3. Disable Caching

There may be cases where you do not want your Spring Boot application to cache application state with [Spring's Cache Abstraction](#) using Apache Geode. In certain cases, you may be using another Spring supported caching provider, such as Redis, to cache and manage your application state, while, even in other cases, you may not want to use Spring's Cache Abstraction at all.

Either way, you can specifically call out your Spring Cache Abstraction provider using the

`spring.cache.type` property in `application.properties`, as follows:

Use Redis as the Spring Cache Abstraction Provider

```
#application.properties  
spring.cache.type=redis  
...
```

If you prefer not to use Spring's Cache Abstraction to manage your Spring Boot application's state at all, then do the following:

Disable Spring's Cache Abstraction

```
#application.properties  
spring.cache.type=none  
...
```

See Spring Boot [docs](#) for more details.



It is possible to include multiple providers on the classpath of your Spring Boot application. For instance, you might be using Redis to cache your application's state while using Apache Geode as your application's persistent data store (*System of Record*).



Spring Boot does not properly recognize `spring.cache.type=[gemfire|geode]` even though Spring Boot for Apache Geode is setup to handle either of these property values (i.e. either “gemfire” or “geode”).

Chapter 10. Data Access with GemfireTemplate

There are several ways to access data stored in Apache Geode.

For instance, developers may choose to use the [Region API](#) directly. If developers are driven by the application's domain context, they might choose to leverage the power of [Spring Data Repositories](#) instead.

While using the *Region* API directly offers flexibility, it couples your application to Apache Geode, which is usually undesirable and unnecessary. While using Spring Data *Repositories* provides a very powerful and convenient abstraction, you give up flexibility provided by a lower level API.

A good compromise is to use the *Template* pattern. Indeed, this pattern is consistently and widely used throughout the entire Spring portfolio.

For example, there is the [JdbcTemplate](#) and [JmsTemplate](#), which are provided by the core Spring Framework.

Other Spring Data modules, such as Spring Data Redis, offer the [RedisTemplate](#), and Spring Data for Apache Geode (SDG) offers the [GemfireTemplate](#).

The [GemfireTemplate](#) provides a highly consistent and familiar API to perform data access operations on Apache Geode cache [Regions](#).

[GemfireTemplate](#) offers:

1. Simple, consistent and convenient data access API to perform CRUD and basic query operations on cache Regions.
2. Use of Spring Framework's consistent, data access [Exception Hierarchy](#).
3. Automatic enlistment in the presence of local, cache transactions.
4. Protection from [Region API](#) breaking changes.

Given these conveniences, Spring Boot for Apache Geode (SBDG) will auto-configure [GemfireTemplate](#) beans for each Region present in the Apache Geode cache.

Additionally, SBDG is careful not to create a [GemfireTemplate](#) if the user has already declared a [GemfireTemplate](#) bean in the Spring [ApplicationContext](#) for a given Region.

10.1. Explicitly Declared Regions

Given an explicitly declared Region bean definition:

```

@Configuration
class GemFireConfiguration {

    @Bean("Example")
    ClientRegionFactoryBean<?, ?> exampleRegion (GemFireCache gemfireCache) {
        // ...
    }
}

```

SBDG will automatically create a **GemfireTemplate** bean for the "Example" Region using a bean name "exampleTemplate". SBDG will name the **GemfireTemplate** bean after the Region by converting the first letter in the Region's name to lowercase and appending the word "Template" to the bean name.

In a managed Data Access Object (DAO), I can inject the Template, like so:

```

@Repository
class ExampleDataAccessObject {

    @Autowired
    @Qualifier("exampleTemplate")
    private GemfireTemplate exampleTemplate;

}

```

It's advisable, especially if you have more than 1 Region, to use the **@Qualifier** annotation to qualify which **GemfireTemplate** bean you are specifically referring as demonstrated above.

10.2. Entity-defined Regions

SBDG auto-configures **GemfireTemplate** beans for Entity-defined Regions.

Given the following entity class:

```

@Region("Customers")
class Customer {
    // ...
}

```

And configuration:

```

@Configuration
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
class GeodeConfiguration {
    // ...
}

```

SBDG auto-configures a `GemfireTemplate` bean for the "Customers" Region named "customersTemplate", which you can then inject into an application component:

```
@Service
class CustomerService {

    @Bean
    @Qualifier("customersTemplate")
    private GemfireTemplate customersTemplate;

}
```

Again, be careful to qualify the `GemfireTemplate` bean injection if you have multiple Regions, whether declared explicitly or implicitly, such as when using the `@EnableEntityDefineRegions` annotation.

10.3. Caching-defined Regions

SBDG auto-configures `GemfireTemplate` beans for Caching-defined Regions.

When you are using Spring Framework's `Cache Abstraction` backed by Apache Geode, 1 of the requirements is to configure Regions for each of the caches specified in the `Caching Annotations` of your application service components.

Fortunately, SBDG makes enabling and configuring caching easy and `automatic` out-of-the-box.

Given a cacheable application service component:

```
@Service
class CacheableCustomerService {

    @Bean
    @Qualifier("customersByNameTemplate")
    private GemfireTemplate customersByNameTemplate;

    @Cacheable("CustomersByName")
    public Customer findBy(String name) {
        return toCustomer(customersByNameTemplate.query("name = " + name));
    }

}
```

And configuration:

```

@Configuration
@EnableCachingDefinedRegions
class GemFireConfiguration {

    @Bean
    public CustomerService customerService() {
        return new CustomerService();
    }
}

```

SBDG auto-configures a `GemfireTemplate` bean named "customersByNameTemplate" used to perform data access operations on the "CustomersByName" (`@Cacheable`) Region, which you can inject into any managed application component, as shown above.

Again, be careful to qualify the `GemfireTemplate` bean injection if you have multiple Regions, whether declared explicitly or implicitly, such as when using the `@EnableCachingDefineRegions` annotation.



There are certain cases where autowiring (i.e. injecting) `GemfireTemplate` beans auto-configured by SBDG for Caching-defined Regions into your application components will not always work! This has to do with the Spring Container bean creation process. In those case you may need to lazily lookup the `GemfireTemplate` as needed, using `applicationContext.getBean("customersByNameTemplate", GemfireTemplate.class)`. This is certainly not ideal but works when autowiring does not.

10.4. Native-defined Regions

SBDG will even auto-configure `GemfireTemplate` beans for Regions defined using Apache Geode native configuration metadata, such as `cache.xml`.

Given the following Apache Geode native `cache.xml`:

```

<?xml version="1.0" encoding="UTF-8"?>
<client-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="Example" refid="LOCAL"/>

</client-cache>

```

And Spring configuration:

```
@Configuration
@EnableGemFireProperties(cacheXmlFile = "cache.xml")
class GemFireConfiguration {
    // ...
}
```

SBDG will auto-configure a `GemfireTemplate` bean named "exampleTemplate" after the "Example" Region defined in `cache.xml`. This Template can be injected like any other Spring managed bean:

```
@Service
class ExampleService {

    @Autowired
    @Qualifier("exampleTemplate")
    private GemfireTemplate exampleTemplate;

}
```

The same rules as above apply when multiple Regions are present.

10.5. Template Creation Rules

Fortunately, SBDG is careful not to create a `GemfireTemplate` bean for a Region if a Template by the same name already exists. For example, if you defined and declared the following configuration:

```
@Configuration
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
class GemFireConfiguration {

    @Bean
    public GemfireTemplate customersTemplate(GemFireCache cache) {
        return new GemfireTemplate(cache.getRegion("/Customers"));
    }

}
```

Using our same Customers class, as above:

```
@Region("Customers")
class Customer {
    // ...
}
```

Because you explicitly defined the "customersTemplate" bean, SBDG will not create a Template for the "Customers" Region automatically. This applies regardless of how the Region was created, whether using `@EnableEntityDefinedRegions`, `@EnableCachingDefinedRegions`, declaring Regions

explicitly or defining Regions natively.

Even if you name the Template differently from the Region for which the Template was configured, SBDG will conserve resources and not create the Template.

For example, suppose you named the `GemfireTemplate` bean, "vipCustomersTemplate", even though the Region name is "Customers", based on the `@Region` annotated `Customer` class, which specified Region "Customers".

With the following configuration, SBDG is still careful not to create the Template:

```
@Configuration
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
class GeodeConfiguration {

    @Bean
    public GemfireTemplate vipCustomersTemplate(GemFireCache cache) {
        return new GemfireTemplate(cache.getRegion("/Customers"));
    }
}
```

SBDG identifies that your "vipCustomersTemplate" is the Template used with the "Customers" Region and SBDG will not create the "customersTemplate" bean, which would result in 2 `GemfireTemplate` beans for the same Region.



The name of your Spring bean defined in JavaConfig is the name of the method if the Spring bean is not explicitly named using the `name` (or `value`) attribute of the `@Bean` annotation.

Chapter 11. Spring Data Repositories

Using Spring Data Repositories with Apache Geode makes short work of data access operations when using Apache Geode as your System of Record (SOR) to persist your application's state.

[Spring Data Repositories](#) provides a convenient and highly powerful way to define basic CRUD and simple query data access operations easily just by specifying the contract of those data access operations in a Java interface.

Spring Boot for Apache Geode *auto-configures* the Spring Data for Apache Geode [Repository extension](#) when either is declared on your application's classpath. You do not need to do anything special to enable it. Simply start coding your application-specific Repository interfaces and the way you go.

For example:

Define a `Customer` class to model customers and map it to the Apache Geode "Customers" Region using the SDG `@Region` mapping annotation:

Customer entity class

```
package example.app.books.model;

@Region("Customers")
class Customer {

    @Id
    private Long id;

    private String name;

}
```

Declare your *Repository* (a.k.a. [Data Access Object \(DAO\)](#)) for `Customers`...

CustomerRepository for persisting and accessing Customers

```
package example.app.books.repo;

interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findByLastNameLikeOrderByLastNameDescFirstNameAsc(String
customerLastNameWildcard);

}
```

Then use the `CustomerRepository` in an application service class:

Inject and use the `CustomerRepository`

```
package example.app;

@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
class SpringBootApacheGeodeClientCacheApplication {

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

    @Bean
    ApplicationRunner runner(CustomerRepository customerRepository) {

        // Matches Williams, Wilson, etc.
        List<Customer> customers =

customerRepository.findByLastNameLikeOrderByLastNameDescFirstNameAsc("Wil%");

        // process the list of matching customers...
    }
}
```

Again, see Spring Data Commons' [Repositories abstraction](#) in general, and Spring Data for Apache Geode's [Repositories extension](#) in particular, for more details.

Chapter 12. Function Implementations & Executions

This chapter is about using Apache Geode in a Spring context for distributed compute use cases.

12.1. Background

Distributed processing, particularly in conjunction with data access and mutation operations, is a very effective and efficient use of clustered computing resources. This is along the same lines as [MapReduce](#).

A naively conceived query returning potentially hundreds of thousands, or even millions of rows of data in a result set back to the application that queried and requested the data can be very costly, especially under load. Therefore, it is typically more efficient to move the processing and computations on the predicated data set to where the data resides, perform the required computations, summarize the results and then send the reduced data set back to the client.

Additionally, when the computations are handled in parallel, across the cluster of computing resources, the operation can be performed much faster. This typically involves intelligently organizing the data using various partitioning (a.k.a. sharding) strategies to uniformly balance the data set across the cluster.

Well, Apache Geode addresses this very important application concern in its [Function Execution](#) framework.

Spring Data for Apache Geode [builds](#) on this Function Execution framework by enabling developers to [implement](#) and [execute](#) Apache Geode Functions using a very simple POJO-based, annotation configuration model.



See [here](#) for the difference between Function implementation & executions.

Taking this 1 step further, Spring Boot for Apache Geode *auto-configures* and enables both Function implementation and execution out-of-the-box. Therefore, you can immediately begin writing Functions and invoking them without having to worry about all the necessary plumbing to begin with. You can rest assured that it will just work as expected.

12.2. Applying Functions

Earlier, when we talked about [caching](#), we described a `FinancialLoanApplicationService` class that could process eligibility when a `Person` applied for a financial loan.

This can be a very resource intensive & expensive operation since it might involve collecting credit and employment history, gathering information on existing, outstanding/unpaid loans, and so on and so forth. We applied caching in order to not have to recompute, or redetermine eligibility every time a loan office may want to review the decision with the customer.

But what about the process of computing eligibility in the first place?

Currently the application's `FinancialLoanApplicationService` class seems to be designed to fetch the data and perform the eligibility determination in place. However, it might be far better to distribute the processing and even determine eligibility for a larger group of people all at once, especially when multiple, related people are involved in a single decision, as is typically the case.

We implement an `EligibilityDeterminationFunction` class using SDG very simply as:

Function implementation

```
@Component
class EligibilityDeterminationFunction {

    @GemfireFunction(HA = true, hasResult = true, optimizeForWrite=true)
    public EligibilityDecision determineEligibility(FunctionContext functionContext,
        Person person, Timespan timespan) {
        // ...
    }
}
```

Using the SDG `@GemfireFunction` annotation, it is easy to implement our Function as a POJO method. SDG handles registering this POJO method as a proper Function with Apache Geode appropriately.

If we now want to call this Function from our Spring Boot, `ClientCache` application, then we simply define a Function Execution interface with a method name matching the Function name, and targeting the execution on the "`EligibilityDecisions`" Region:

Function execution

```
@OnRegion("EligibilityDecisions")
interface EligibilityDeterminationExecution {

    EligibilityDecision determineEligibility(Person person, Timespan timespan);

}
```

We can then inject the `EligibilityDeterminationExecution` into our `FinancialLoanApplicationService` like any other object/Spring bean:

```
@Service
class FinancialLoanApplicationService {

    private final EligibilityDeterminationExecution execution;

    public LoanApplicationService(EligibilityDeterminationExecution execution) {
        this.execution = execution;
    }

    @Cacheable("EligibilityDecisions")
    EligibilityDecision processEligibility(Person person, Timespan timespan) {
        return this.execution.determineEligibility(person, timespan);
    }
}
```

Just like caching, no addition configuration is required to enable and find your application Function implementations and executions. Simply build and run. Spring Boot for Apache Geode handles the rest.



It is common to implement and register your application Functions on the server and execute them from the client.

Chapter 13. Continuous Query

Arguably, the most invaluable of applications are those that can process a stream of events as they happen, and intelligently react in near real-time to the countless changes in the data over time. The most useful of frameworks are those that can make processing a stream of events as they happen, as easy as possible.

Spring Boot for Apache Geode does just that, without users having to perform any complex setup or configure any necessary infrastructure components to enable such functionality. Developers can simply define the criteria for the data they are interested in and implement a handler to process the stream of events as they occur.

Apache Geode make defining criteria for data of interests easy when using [Continuous Query \(CQ\)](#). With CQ, you can express the criteria matching the data of interests using a query predicate. Apache Geode implements the [Object Query Language \(OQL\)](#) for defining and executing queries. OQL is not unlike SQL, and supports projections, query predicates, ordering and aggregates. And, when used in CQs, they execute continuously, firing events when the data changes in such ways as to match the criteria expressed in the query predicate.

Spring Boot for Apache Geode combines the ease of expressing interests in data using an OQL query statement with implementing the listener handler callback, in 1 easy step.

For example, suppose we want to perform some follow up action anytime a customer's financial loan application is either approved or denied.

First, the application model for our `EligibilityDecision` class might look something like:

EligibilityDecision class

```
@Region("EligibilityDecisions")
class EligibilityDecision {

    private final Person person;

    private Status status = Status.UNDETERMINED;

    private final Timespan timespan;

    enum Status {

        APPROVED,
        DENIED,
        UNDETERMINED,

    }

}
```

Then, we can implement and declare our CQ event handler methods to be notified when a decision is either APPROVED or DENIED:

```

@Component
class EligibilityDecisionPostProcessor {

    @ContinuousQuery(name = "ApprovedDecisionsHandler",
        query = "SELECT decisions.*
                FROM /EligibilityDecisions decisions
                WHERE decisions.getStatus().name().equalsIgnoreCase('APPROVED')")
    public void processApprovedDecisions(CqEvent event) {
        // ...
    }

    @ContinuousQuery(name = "DeniedDecisionsHandler",
        query = "SELECT decisions.*
                FROM /EligibilityDecisions decisions
                WHERE decisions.getStatus().name().equalsIgnoreCase('DENIED')")
    public void processDeniedDecisions(CqEvent event) {
        // ...
    }
}

```

Thus, anytime eligibility is processed and a decision as been made, either approved or denied, our application will get notified, and as an application developer, you are free to code your handler and respond to the event anyway you like. And, because our Continuous Query handler class is a component, or bean in the Spring [ApplicationContext](#), you can auto-wire any other beans necessary to carry out the application's intended function.

This is not unlike Spring's [Annotation-driven listener endpoints](#) used in (JMS) message listeners/handlers, except in Spring Boot for Apache Geode, you do not need to do anything special to enable this functionality. Just declare the [@ContinuousQuery](#) annotation on any POJO method and off you go.

Chapter 14. Using Data

One of the most important tasks during development is ensuring your Spring Boot application handles data correctly. In order to verify the accuracy, integrity and availability of your data, your application needs data to work with.

For those already familiar with Spring Boot's support for [SQL database initialization](#), the approach when using Apache Geode should be easy to understand.

Apache Geode provides built-in support, similar in function to Spring Boot's SQL database initialization, by using:

- *Gfsh*'s [import/export](#) data commands.
- [Snapshot Service](#)
- [Persistence](#) with [Disk Storage](#)

For example, by enabling Persistence with Disk Storage, you could [backup and restore](#) persistent [DiskStore](#) files from one cluster to another.

Alternatively, using Apache Geode's *Snapshot Service*, you can export data contained in targeted [Regions](#) from one cluster during shutdown and import the data into another cluster on startup. The *Snapshot Service* allows you to filter data while its being imported and exported.

Finally, Apache Geode Shell (*Gfsh*) commands can be used to [export data](#) and [import data](#).



Spring Data for Apache Geode (SDG) contains dedicated support for [Persistence](#) and the [Snapshot Service](#).

In all cases, the files generated by *persistence*, the *Snapshot Service* and *Gfsh*'s [export](#) command are in a proprietary, binary format.

Furthermore, none of these approaches are as convenient as Spring Boot's database initialization automation. Therefore, Spring Boot for Apache Geode (SBDG) offers support to import data from JSON into Apache Geode as PDX.

Unlike Spring Boot, SBDG offers support to export data as well. Data is imported and exported in JSON format, by default.



SBDG does not provide an equivalent to Spring Boot's [schema.sql](#) file. The best way to define the data structures (i.e. [Regions](#)) managing your data is with SDG's Annotation-based configuration support for defining cache [Regions](#) from your application's [entity classes](#) or indirectly from Spring and JSR-107, JCache [caching annotations](#).



Refer to SBDG's [documentation](#) on the same.



While this feature has utility and many edge cases were thought through and tested thoroughly, there are still some limitations that need to be ironed out. See [Issue-82](#) and [Issue-83](#) for more details. The Spring team strongly recommends that this feature only be used for development and testing purposes.

14.1. Importing Data

You can import data into a **Region** by defining a JSON file containing the JSON object(s) you wish to load. The JSON file must follow the naming convention below and be placed in the root of your application classpath:

`data-<regionName>.json`



`<regionName>` refers to the lowercase "name" of the **Region** as defined by `Region.getName()`.

For example, if you have a **Region** named "Orders", then you would create a JSON file called `data-orders.json` and place it in the root of your application classpath (e.g. in `src/test/resources`).

Create JSON files for each **Region** implicitly defined (e.g. by using `@EnableEntityDefinedRegions`) or explicitly defined (i.e. with `ClientRegionFactoryBean` in `JavaConfig`) in your Spring Boot application configuration that you want to load with data.

The JSON file containing JSON data for *Orders* might appear as follows:

```
[{
  "@type": "example.app.pos.model.PurchaseOrder",
  "id": 1,
  "lineItems": [
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Apple iPad Pro",
        "price": 1499.00,
        "category": "SHOPPING"
      },
      "quantity": 1
    },
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Apple iPhone 11 Pro Max",
        "price": 1249.00,
        "category": "SHOPPING"
      },
      "quantity": 2
    }
  ]
}, {
  "@type": "example.app.pos.model.PurchaseOrder",
  "id": 2,
  "lineItems": [
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Starbucks Vente Carmel Macchiato",
        "price": 5.49,
        "category": "SHOPPING"
      },
      "quantity": 1
    }
  ]
}]
```

The application entity classes matching the JSON data might look something like:

```
@Region("Orders")
class PurchaseOrder {

    @Id
    Long id;

    List<LineItem> lineItems;
}

class LineItem {

    Product product;
    Integer quantity;
}

@Region("Products")
class Product {

    String name;
    Category category;
    BigDecimal price;
}
```

As seen above, the object model and corresponding JSON can be arbitrarily complex with a hierarchy of objects having complex types.

14.1.1. JSON metadata

You will notice a few other details contained in the object model and JSON shown above.

The `@type` metadata field

First, we declared an `@type` JSON metadata field. This field does not map to any specific field or property of the application domain model class (e.g. `PurchaseOrder`). Rather, it tells the framework and Apache Geode's JSON/PDX converter the type of object the JSON data would map to if you were to request an object (i.e. by calling `PdxInstance.getObject()`).

For example:

```
@Repository
class OrdersRepository {

    @Resource(name = "Orders")
    Region<Long, PurchaseOrder> orders;

    PurchaseOrder findBy(Long id) {

        Object value = this.orders.get(id);

        return value instanceof PurchaseOrder ? (PurchaseOrder) value
            : value instanceof PdxInstance ? ((PdxInstance) value).getObject()
            : null;
    }
}
```

Basically, the `@type` JSON metadata field informs the `PdxInstance.getObject()` method about the type of Java object the JSON object will map to. Otherwise, the `PdxInstance.getObject()` method would silently return a `PdxInstance`.

It is possible for Apache Geode's PDX serialization framework to return a `PurchaseOrder` from `Region.get(key)` as well, but it depends on the value of PDX's `read-serialized`, cache-level configuration setting, among other factors.



When JSON is imported into a `Region` as PDX, the `PdxInstance.getClassName()` does not refer to a valid Java class. It is `JSONFormatter.JSON_CLASSNAME`. As a result, `Region` data access operations, such as `Region.get(key)`, return a `PdxInstance` and not a Java object.



You may need to proxy `Region` "read" data access operations (e.g. `Region.get(key)`) by setting the SBDG property `spring.boot.data.gemfire.cache.region.advice.enabled` to `true`. When this property is set, `Regions` are proxied to wrap a `PdxInstance` in a `PdxInstanceWrapper` in order to appropriately handle the `PdxInstance.getObject()` call in your application code.

The `id` field & `@identifier` metadata field

The top-level objects in your JSON must have an identifier, such as an "id" field. This identifier is used as the object's (or `PdxInstance`s) identity and "key" when stored in the `Region` (e.g. `Region.put(key, object)`).

You will have noticed the the JSON for the `Orders` above declared an "id" field as the identifier:

PurchaseOrder identifier ("id")

```
[{
  "@type": "example.app.pos.model.PurchaseOrder",
  "id": 1,
  ...
}]
```

This follows the same convention used in Spring Data. Typically, Spring Data mapping infrastructure looks for a POJO field or property annotated with `@Id`. If no field or property is annotated with `@Id`, then the framework falls back to searching for a field or property named "id".

In Spring Data for Apache Geode (SDG), this `@Id` annotated, or "id" named field or property is used as the identifier, and as the key for the object when storing it into a `Region`.

However, what happens when an object, or entity does not have a surrogate id defined? Perhaps the application domain model class is appropriately and simply using "natural" identifiers, which is quite common in practice.

Consider a `Book` class defined as follows:

Book class

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

    Author author;

    @Id
    ISBN isbn;

    LocalDate publishedDate;

    String title;

}
```

As declared in the `Book` class above, the identifier for `Book` is its `ISBN` since the `isbn` field was annotated with Spring Data's `@Id` mapping annotation. However, we cannot know this by searching for an `@Id` annotation in JSON.

You might be tempted to argue that if the `@type` metadata field is set, we would know the class type and could load the class definition to learn about the identifier. That is all fine until the class is not actually on the application classpath in the first place. This is one of the reasons why SBDG's JSON support serializes JSON to Apache Geode's PDX format. There might not be a class definition, which would lead to a `NoClassDefFoundError` or `ClassNotFoundException`.

So, what then?

In this case, SBDG allows you to declare the `@identifier` JSON metadata field to inform the framework what to use as the identifier for the object.

For example:

Using "@identifer"

```
{
  "@type": "example.app.books.model.Book",
  "@identifier": "isbn",
  "author": {
    "id": 1,
    "name": "Josh Long"
  },
  "isbn": "978-1-449-374640-8",
  "publishedDate": "2017-08-01",
  "title": "Cloud Native Java"
}
```

Here, the `@identifier` JSON metadata field informs the framework that the "isbn" field is the identifier for a `Book`.

14.1.2. Conditionally Importing Data

While the Spring team recommends that users should only use this feature when developing and testing their Spring Boot applications with Apache Geode, a user may occasionally use this feature in production.

Users might use this feature in production to preload a (REPLICATE) Region with "reference" data. Reference data is largely static, infrequently changing and non-transactional. Preloading reference data is particularly useful in caching use cases, where you want to "warm" the cache.

When using this feature for development and testing purposes, you can simply put your `Region` specific JSON files in `src/test/resources`. This ensures the files will not be included in your application artifact (e.g. JAR, WAR) when deployed to production.

However, if you must use this feature to preload data in your production environment, then you can still "conditionally" load data from JSON. To do so configure the `spring.boot.data.gemfire.cache.data.import.active-profiles` property set to the Spring profile(s) that must be active for the import to take effect.

For example:

Conditional Importing JSON

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.active-profiles=DEV, QA
```

In order for import to have an effect in this example, you must specifically set the `spring.profiles.active` property to 1 of the valid, "active-profiles" listed in the import property (e.g. `QA`). Only 1 needs to match.



There are many ways to conditionally build application artifacts. Some users might prefer to handle this concern in their Gradle or Maven builds.

14.2. Exporting Data

Certain data stored in your application's **Regions** may be sensitive or confidential and keeping the data secure is of the utmost concern and priority. Therefore, exporting data is **disabled** by default.

However, if you are using this feature for development and testing purposes then enabling the *export* capability may be useful to move data from 1 environment to another. For example, if your QA team finds a bug in the application using a particular data set, then they can *export* the data and pass it back to the development team to *import* in their local development environment to help debug the issue.

To enable *export*, set the `spring.boot.data.gemfire.cache.data.export.enabled` property to `true`:

Enable Export

```
# Spring Boot application.properties  
  
spring.boot.data.gemfire.cache.data.export.enabled=true
```

SBDG is careful to *export* data to JSON in a format that Apache Geode expects on *import* and includes things such as `@type` metadata fields.



The `@identifier` metadata field is not generated automatically. While it is possible for POJOs stored in a **Region** to include an `@identifier` metadata field when exported to JSON it is not possible when the **Region** value is a **PdxInstance** that did not originate from JSON. In this case, you must manually ensure the **PdxInstance** includes an `@identifier` metadata field before it is exported to JSON if necessary (e.g. `Book.isbn`). This is only necessary if your entity classes do not declare an explicit identifier field, such as with the `@Id` mapping annotation, or do not have an "id" field. This scenario can also occur when inter-operating with native clients that model the application domain objects differently, then serialize the objects using PDX storing them in **Regions** on the server that are then later consumed by your Spring Boot application.



It may be necessary to set the `-Dgemfire.disableShutdownHook` JVM System property to `true` before your Spring Boot application starts up when using Export. Unfortunately, this Java Runtime shutdown hook is registered and enabled in Apache Geode by default, which results in the cache and **Regions** being closed before the SBDG Export functionality can "export the data", thereby resulting in a `CacheClosedException`. SBDG **makes a best effort** to disable the Apache Geode shutdown hook when export is enabled, but it is at the mercy of the JVM **ClassLoader** since Apache Geode's JVM shutdown hook **registration** is declared in a `static` initializer.

14.3. Import/Export API Extensions

The API in SBDG for Import/Export functionality is separated into the following concerns:

- Data Format
- Resource Resolving
- Resource Reading
- Resource Writing

By breaking each of these functions apart into separate concerns, it affords a developer the ability to customize each aspect of the Import/Export functions.

For example, you could import XML from the filesystem and then export JSON to a REST-based Web Service. By default, SBDG imports JSON from the classpath and exports JSON to the filesystem.

However, not all environments expose the filesystem, such as cloud environments like PCF. Therefore, giving users control over each aspect of import/export process is essential for performing the functions in any environment.

14.3.1. Data Format

The primary interface to import data into a `Region` is the `CacheDataImporter`.

`CacheDataImporter` is a `@FunctionalInterface` extending Spring's `BeanPostProcessor` interface to trigger the import of data after the `Region` has been initialized.

The interface is defined as:

`CacheDataImporter`

```
interface CacheDataImporter extends BeanPostProcessor {  
    Region importInto(Region region);  
}
```

The `importInto(..)` method can be coded to handle any data format (JSON, XML, etc) you prefer. Simply register a bean implementing the `CacheDataImporter` interface in the Spring container and the importer will do its job.

On the flip-side, the primary interface to export data from a `Region` is the `CacheDataExporter`.

`CacheDataExporter` is a `@FunctionalInterface` extending Spring's `DestructionAwareBeanPostProcessor` interface to trigger the export of data before the `Region` is destroyed.

The interface is defined as:

CacheDataExporter

```
interface CacheDataExporter extends DestructionAwareBeanPostProcessor {  
    Region exportFrom(Region region);  
}
```

The `exportFrom(..)` method can be coded to handle any data format (JSON, XML, etc) you prefer. Simply register a bean implementing the `CacheDataExporter` interface in the Spring container and the exporter will do its job.

For convenience when you want to implement both import and export functionality, SBDG provides the `CacheDataImporterExporter` interface, which extends both `CacheDataImporter` and `CacheDataExporter`.

CacheDataImporterExporter

```
interface CacheDataImporterExporter extends CacheDataExporter, CacheDataImporter { }
```

For support, SBDG also provides the `AbstractCacheDataImporterExporter` abstract base class to simplify the implementation of your importer/exporter.

Lifecycle Management

Sometimes it is necessary to control precisely when data is imported or exported.

This is especially true on import since different `Regions` maybe collocated or tied together via a cache callback like a `CacheListener`. In these cases, the other `Region` may need to exist before the import on the dependent `Region` proceeds, particularly if the dependencies were loosely defined.

Another case when controlling the import is important is when you are using SBDG's `@EnableClusterAware` annotation to push configuration metadata from the client to the cluster in order to define server-side `Regions` matching the client-side `Regions`, especially client `Regions` targeted for import. The matching `Regions` on the server-side must exist before data is imported into client (PROXY) `Regions`.

In all cases, SBDG provides the `LifecycleAwareCacheDataImporterExporter` class to wrap your `CacheDataImporterExporter` implementation. This class implements Spring's `SmartLifecycle` interface.

By implementing the `SmartLifecycle` interface, it allows you to control which `phase` of the Spring container the import occurs. As such SBDG exposes two more properties to control the lifecycle:

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.lifecycle=[EAGER|LAZY]
spring.boot.data.gemfire.cache.data.import.phase=1000000
```

EAGER acts immediately, after the Region is initialized (the default behavior). **LAZY** delays the import until the **start()** method is called, which is invoked according to the **phase**, thereby ordering the import relative to other "lifecycle-aware" components registered in the Spring container.

To make your **CacheDataImporterExporter** "lifecycle-aware" simply do:

```
@Configuration
class MyApplicationConfiguration {

    @Bean
    CacheDataImporterExporter importerExporter() {
        return new LifecycleAwareCacheDataImporterExporter(new
MyCacheDataImporterExporter());
    }
}
```

14.3.2. Resource Resolution

Resolving resources used for import and export results in the creation of a Spring **Resource** handle.

Resource resolution is a vital step to qualify a resource, especially if the resource requires special logic or permissions to access it. In this case, specific **Resource** handles can be returned and used by the *reader* and *writer* of the **Resource** as is appropriate for import or export operation.

SBDG encapsulates the algorithm for resolving **Resources** in the **ResourceResolver** (**Strategy**) interface:

ResourceResolver

```
@FunctionalInterface
interface ResourceResolver {

    Optional<Resource> resolve(String location);

    default Resource required(String location) {
        // ...
    }
}
```

Additionally, SBDG provides the **ImportResourceResolver** and **ExportResourceResolver** marker interfaces along with the **AbstractImportResourceResolver** and **AbstractExportResourceResolver**

abstract base classes for implementing resource resolution logic used by both import and export operations, for your convenience.

If you wish to customize the resolution of **Resources** used for import and/or export, your **CacheDataImporterExporter** implementation can extend the **ResourceCapableCacheDataImporterExporter** abstract base class, which provides the aforementioned interfaces and base classes.

As stated above, SBDG resolves resources on import from the classpath and resources on export to the filesystem.

It is easy to customize this behavior simply by providing an implementation of either or both the **ImportResourceResolver** and **ExportResourceResolver** interfaces and declare instances as beans in the Spring context:

Import & Export ResourceResolver beans

```
@Configuration
class MyApplicationConfiguration {

    @Bean
    ImportResourceResolver importResourceResolver() {
        return new MyImportResourceResolver();
    }

    @Bean
    ExportResourceResolver exportResourceResolver() {
        return new MyExportResourceResolver();
    }
}
```



If you need to customize the resource resolution process per location (or **Region**) on import or export, then you could use the [Composite Software Design Pattern](#).

Customize Default Resource Resolution

If you are content with the provided defaults, but want to target specific locations on the classpath or filesystem used by the import or export, then SBDG additionally provides the following properties:

Import/Export Resource Location Properties

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.resource.location=...
spring.boot.data.gemfire.cache.data.export.resource.location=...
```

The properties accept any valid resource string as specified in the Spring [documentation](#) (See **Table 10. Resource strings**).

This means even though the import defaults from the classpath, it is simple to change the location from classpath to filesystem, or even network (e.g. https://) simply by changing the *prefix* (or *protocol*).

Of course, import/export resource location properties can refer to other properties via property placeholders, but SBDG further allows users to use SpEL inside the property values.

For example:

Using SpEL

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.resource.location=\

https://#{#env['user.name']}:{someBean.lookupPassword(#env['user.name'])}@#{host}:{port}/cache/#{#regionName}/data/import
```

The import resource location in this case refers to a rather sophisticated resource string using a complex SpEL expression.

Out-of-the-box, SBDG populates the SpEL `EvaluationContext` with 3 sources of information:

- Access to the Spring `BeanFactory`
- Access to the Spring `Environment`
- Access to the current `Region`

Simple Java System properties or environment variables can be accessed with the expression:

```
#{propertyName}
```

For more complex property names (e.g. properties using dot notation, such as the `user.home` Java System property), users can access these properties directly from the `Environment` using map style syntax as follows:

```
#{#env['property.name']}
```

The `#env` variable is set in the SpEL `EvaluationContext` to the Spring `Environment`.

Because the SpEL `EvaluationContext` is evaluated with the Spring `ApplicationContext` as the root object, you also have access to the beans declared and registered in the Spring context and can invoke methods on them, as shown above with `someBean.lookupPassword(..)`. "someBean" must be the name of the bean as declared/registered in the Spring context.



Be careful when accessing beans declared in the Spring context with SpEL, particularly when using `EAGER` import as it may force those beans to be eagerly (or even, prematurely) initialized.

SBDG also sets the `#regionName` variable in the `EvaluationContext` to the name of the `Region`, as determined by `Region.getName()`, targeted for import/export.

This allows you to not only change the location of the resource but also change the resource name (e.g. filename).

For example:

Using `#regionName`

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.export.resource.location=\
    file://${#env['user.home']}/gemfire/cache/data/custom-filename-for-
    ${#regionName}.json
```



By default, the exported file is stored in the working directory (i.e. `System.getProperty("user.dir")`) of the Spring Boot application process.



See the Spring [documentation](#) for more information on SpEL.

14.3.3. Reading & Writing Resources

The Spring `Resource` handle specifies the location of a resource, not how to read or write it. Even the Spring `ResourceLoader`, which is an interface for "loading" `Resources`, does not specifically read or write any content to the `Resource`.

As such, SBDG separates these concerns into two interfaces: `ResourceReader` and `ResourceWriter`, respectively. The design follows the same pattern used by Java's `InputStream/OutputStream` and `Reader/Writer` classes in the `java.io` package.

The interfaces are basically defined as:

ResourceReader

```
@FunctionalInterface
interface ResourceReader {

    byte[] read(Resource resource);

}
```

And...

```
@FunctionalInterface
interface ResourceWriter {

    void write(Resource resource, byte[] data);

}
```

Both of interfaces provide additional methods to *compose* readers and writers, much like Java's own `Consumer` and `Function` interfaces in the `java.util.function` package. If a particular reader or writer is used in a composition and is unable to handle the given `Resource`, then it should throw a `UnhandledResourceException` to allow the next reader or writer in the composition to try and read from or write to the `Resource`.

Of course, the reader or writer are free to throw a `ResourceReadException` or `ResourceWriteException` to break the chain of reader and writer invocations in the composition.

To override the default export/import reader and writer used by SBDG out-of-the-box, simply implement the `ResourceReader` and/or `ResourceWriter` interfaces as appropriate and declare instances of these classes as beans in the Spring context:

Custom `ResourceReader` & `ResourceWriter` beans

```
@Configuration
class MyApplicationConfiguration {

    @Bean
    ResourceReader myResourceReader() {
        return new MyResourceReader()
            .thenReadFrom(new MyOtherResourceReader());
    }

    @Bean
    ResourceWriter myResourceWriter() {
        return new MyResourceWriter();
    }

}
```

Chapter 15. Data Serialization with PDX

Anytime data is overflowed or persisted to disk, transferred between clients and servers, peers in a cluster or between different clusters in a multi-site topology, then all data stored in Apache Geode must be serializable.

To serialize objects in Java, object types must implement the `java.io.Serializable` interface. However, if you have a large number of application domain object types that currently do not implement `java.io.Serializable`, then refactoring hundreds or even thousands of class types to implement `Serializable` would be a tedious task just to store and manage those objects in Apache Geode.

Additionally, it is not just your application domain object types you necessarily need to worry about either. If you used 3rd party libraries in your application domain model, any types referred to by your application domain object types stored in Apache Geode must be serializable too. This type explosion may bleed into class types for which you may have no control over.

Furthermore, Java serialization is not the most efficient format given that meta-data about your types is stored with the data itself. Therefore, even though Java serialized bytes are more descriptive, it adds a great deal of overhead.

Then, along came serialization using Apache Geode's `PDX` format. PDX stands for *Portable Data Exchange*, and achieves 4 goals:

1. Separates type meta-data from the data itself making the bytes more efficient during transfer. Apache Geode maintains a type registry storing type meta-data about the objects serialized using PDX.
2. Supports versioning as your application domain types evolve. It is not uncommon to have old and new applications deployed to production, running simultaneously, sharing data, and possibly using different versions of the same domain types. PDX allows fields to be added or removed while still preserving interoperability between old and new application clients without loss of data.
3. Enables objects stored as PDX bytes to be queried without being de-serialized. Constant de/serialization of data is a resource intensive task adding to the latency of each data request when redundancy is enabled. Since data must be replicated across peers in the cluster to preserve High Availability (HA), and serialized to be transferred, keeping data serialized is more efficient when data is updated frequently since it will likely need to be transferred again in order to maintain consistency in the face of redundancy and availability.
4. Enables interoperability between native language clients (e.g. C/C++/C#) and Java language clients, with each being able to access the same data set regardless from where the data originated.

However, PDX is not without its limitations either.

For instance, unlike Java serialization, PDX does not handle cyclic dependencies. Therefore, you must be careful how you structure and design your application domain object types.

Also, PDX cannot handle field type changes.

Furthermore, while Apache Geode's general [Data Serialization](#) handles [deltas](#), this is not achievable without de-serializing the object bytes since it involves a method invocation, which defeats 1 of the key benefits of PDX, preserving format to avoid the cost of de/serialization.

However, we think the benefits of using PDX greatly outweigh the limitations and therefore have enabled PDX by default when using Spring Boot for Apache Geode.

There is nothing special you need to do. Simply code your types and rest assured that objects of those types will be properly serialized when overflowed/persisted to disk, transferred between clients and servers, or peers in a cluster and even when data is transferred over the WAN when using Apache Geode's multi-site topology.

EligibilityDecision is automatically serialiable without implementing Java Serializable.

```
@Region("EligibilityDecisions")
class EligibilityDecision {
    // ...
}
```



Apache Geode does [support](#) the standard Java Serialization format.

15.1. SDG [MappingPdxSerializer](#) vs. Apache Geode's [ReflectionBasedAutoSerializer](#)

Under-the-hood, Spring Boot for Apache Geode [enables](#) and uses Spring Data for Apache Geode's [MappingPdxSerializer](#) to serialize your application domain objects using PDX.



Refer to the SDG [Reference Guide](#) for more details on the [MappingPdxSerializer](#) class.

The [MappingPdxSerializer](#) offers several advantages above and beyond Apache Geode's own [ReflectionBasedAutoSerializer](#) class.



Refer to Apache Geode's [User Guide](#) for more details about the [ReflectionBasedAutoSerializer](#).

The SDG [MappingPdxSerializer](#) offers the following capabilities:

1. PDX serialization is based on Spring Data's powerful mapping infrastructure and meta-data, as such...
2. Includes support for both [includes](#) and [excludes](#) with [type filtering](#). Additionally, type filters can be implemented using Java's `java.util.function.Predicate` interface as opposed to Apache Geode's limited regex capabilities provided by the [ReflectionBasedAutoSerializer](#) class. By default, [MappingPdxSerializer](#) excludes all types in the following packages: `java`, `org.apache.geode`, `org.springframework` & `com.gemstone.gemfire`.
3. Handles [transient object fields & properties](#) when either Java's `transient` keyword or Spring Data's `@Transient` annotation is used.

4. Handles [read-only object properties](#).
5. Automatically determines the identifier of your entities when you annotate the appropriate entity field or property with Spring Data's [@Id](#) annotation.
6. Allows [o.a.g.pdx.PdxSerializers](#) to be registered in order to [customize the serialization](#) of nested entity field/property types.

Number two above deserves special attention since the [MappingPdxSerializer](#) "excludes" all Java, Spring and Apache Geode types, by default. But, what happens when you need to serialize 1 of those types?

For example, suppose you need to be able to serialize objects of type [java.security.Principal](#). Well, then you can override the excludes by registering an "include" type filter, like so:

```
package example.app;

import java.security.Principal;

@SpringBootApplication
@EnablePdx(serializerBeanName = "myCustomMappingPdxSerializer")
class SpringBootApacheGeodeClientCacheApplication {

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

    @Bean
    MappingPdxSerializer myCustomMappingPdxSerializer() {

        MappingPdxSerializer customMappingPdxSerializer =
            MappingPdxSerializer.newMappginPdxSerializer();

        customMappingPdxSerializer.setIncludeTypeFilters(
            type -> Principal.class.isAssignableFrom(type));

        return customMappingPdxSerializer;
    }
}
```



Normally, you do not need to explicitly declare SDG's [@EnablePdx](#) annotation to enable and configure PDX. However, if you want to override auto-configuration, as we have demonstrated above, then this is what you must do.

Chapter 16. Logging

Apache Geode 1.9.2 was modularized to separate its use of the Apache Log4j API to log output in Geode code from the underlying implementation of logging, which uses Apache Log4j as the logging provider by default.

Prior to 1.9.2, the Apache Log4j API (i.e. `log4j-api`) along with the Apache Log4j provider (i.e. `log4j-core`) were automatically pulled in by Apache Geode core (i.e. `org.apache.geode:geode-core`) thereby making it problematic to change logging providers when using Apache Geode in Spring Boot applications.

However, now, in order to get any log output from Apache Geode whatsoever, Apache Geode requires a logging provider on your Spring Boot application classpath. Consequently, this also means the old Apache Geode `Properties`, e.g. `log-level` no longer have any effect, regardless of whether the property (e.g. `log-level`) is specified in `gemfire.properties`, in `Spring Boot application.properties` or even as a JVM System Property, `-Dgemfire.log-level`.



Refer to Apache Geode's [Documentation](#) for a complete list of valid `Properties`, including the `Properties` used to configure logging.

Unfortunately, this also means the *Spring Data for Apache Geode* (SDG) `@EnableLogging` annotation no longer has any effect on Apache Geode logging either and is the reason it has been [deprecated](#). The reason `@EnableLogging` no longer has any effect on logging is because this annotation's attributes and associated SDG properties indirectly sets the corresponding Apache Geode properties, which again, are useless from Apache Geode 1.9.2 onward.

By way of example, and to make this concrete, **none** of the following approaches have any effect on Apache Geode logging:

Command-line configuration

```
$ java -classpath ...:/path/to/MySpringBootApacheGeodeClientCacheApplication.jar
-Dgemfire.log-level=DEBUG
  example.app.MySpringBootApacheGeodeClientCacheApplication
```

Externalized configuration using Apache Geode `gemfire.properties`

```
# {geode-name} only/specific properties
log-level=INFO
```

Externalized configuration using Spring Boot `application.properties`

```
spring.data.gemfire.cache.log-level=DEBUG
```

Or:

```
spring.data.gemfire.logging.level=DEBUG
```

Java configuration using SDG's `@EnableLogging` annotation

```
@SpringBootApplication
@EnableLogging(logLevel = "DEBUG")
class MySpringBootApacheGeodeClientApplication {
    // ...
}
```

That is to say, none of the approaches above have any effect without the **new** SBDG logging starter.

16.1. Configure Apache Geode Logging

So, how do you configure logging for Apache Geode?

Effectively, 3 things are required to get Apache Geode to log output:

- 1) First, you must declare a logging provider on your Spring Boot application classpath (e.g. *Logback*).
- 2) (optional) Next, you must declare an adapter, or bridge JAR, between Log4j and your logging provider if your declared logging provider is not Apache Log4j.

For example, if you use the SLF4J API to log output from your Spring Boot application along with *Logback* as your logging provider/implementation, then you must include the `org.apache.logging.log4j.log4j-to-slf4j` adapter/bridge JAR dependency as well.

Internally, Apache Geode uses the Apache Log4j API to log output from Geode components. Therefore, you must bridge Log4j to any other logging provider (e.g. *Logback*) that is not Log4j (i.e. `log4j-core`). If you are using Log4j as your logging provider then you do not need to declare an adapter/bridge JAR on your Spring Boot application classpath.

- 3) Finally, you must supply logging provider configuration to configure Loggers, Appenders, log levels, etc.

For example, when using *Logback*, you must provide a `logback.xml` configuration file on your Spring Boot application classpath, or in the filesystem. Alternatively, you can use other means to configure your logging provider and get Apache Geode to log output.



Apache Geode's `geode-log4j` module covers the required configuration for steps 1-3 above and uses Apache Log4j (i.e. `org.apache.logging.log4j:log4j-core`) as the logging provider. The `geode-log4j` module even provides a default, `log4j2.xml` configuration file to configure Loggers, Appenders and log levels for Apache Geode.

If you declare Spring Boot's own `org.springframework.boot:spring-boot-starter-logging` on your application classpath then this will cover Steps 1 and 2 above.

The `spring-boot-starter-logging` dependency declares *Logback* as the logging provider and automatically adapts, or bridges `java.util.logging` (JUL) and Apache Log4j to SLF4J. However, you still need to supply logging provider configuration, such as a `logback.xml` file for *Logback*, to configure logging not only for your Spring Boot application, but also for Apache Geode as well.

SBDG has simplified the setup of Apache Geode logging. Simply declare the `org.springframework.geode:spring-geode-starter-logging` dependency on your Spring Boot application classpath!

Unlike Apache Geode's default Log4j XML configuration file (i.e. `log4j2.xml`), SBDG's provided `logback.xml` configuration file is properly parameterized enabling you to adjust log levels as well as add Appenders.

In addition, SBDG's provided *Logback* configuration uses templates so you can compose your own logging configuration while still "including" snippets from SBDG's provided logging configuration metadata, such as Loggers and Appenders.

16.1.1. Configuring Log Levels

One of the most common logging tasks is to adjust the log-level of one or more Loggers, or the ROOT Logger. However, a user may only want to adjust the log-level for specific components of his/her Spring Boot application, such as for Apache Geode, by setting the log-level for only the Logger that logs Apache Geode events.

SBDG's *Logback* configuration defines 3 Loggers to control the log output from Apache Geode:

Apache Geode Loggers by name

```
<configuration>
  <logger name="com.gemstone.gemfire" level="${spring.boot.data.gemfire.log.level:-
INFO}" />
  <logger name="org.apache.geode" level="${spring.boot.data.gemfire.log.level:-
INFO}" />
  <logger name="org.jgroups" level="${spring.boot.data.gemfire.jgroups.log.level:-
ERROR}" />
</configuration>
```

The `com.gemstone.gemfire` Logger is a legacy Logger covering old GemFire bits still present in Apache Geode for backwards compatibility reasons. This Logger's use should be largely unnecessary.

The `org.apache.geode` Logger is the primary Logger used to control log output from all Apache Geode components during the runtime operation of Apache Geode. Both this Logger and the legacy `com.gemstone.gemfire` Logger default log output to `INFO`.

The `org.jgroups` Logger is used to log output from Apache Geode's message distribution and membership system. Apache Geode uses JGroups for membership and message distribution between peer members (nodes) in the cluster (distributed system). By default, JGroups log messages are logged at `ERROR`.

The log-level for the `com.gemstone.gemfire` and `org.apache.geode` Loggers are configured with the `spring.boot.data.gemfire.log.level` property. The `org.jgroups` Logger is independently configured with the `spring.boot.data.gemfire.jgroups.log.level` property.

The SBDG logging properties can be set on the command-line as JVM System Properties when running your Spring Boot application:

Setting the log-level from the command-line

```
$ java -classpath ...:/path/to/MySpringBootApplication.jar
-Dspring.boot.data.gemfire.log.level=DEBUG
package.to.MySpringBootApplicationClass
```



Setting JVM System Properties using `$ java -jar MySpringBootApplication.jar -Dspring.boot.data.gemfire.log.level=DEBUG` is not supported by the Java Runtime Environment (JRE).

Alternatively, you can configure and control Apache Geode logging in Spring Boot application.properties:

Setting the log-level in application.properties

```
spring.boot.data.gemfire.log.level=DEBUG
```

For backwards compatibility, SBDG additionally supports the old *Spring Data for Apache Geode* (SDG) logging properties as well, using either:

```
spring.data.gemfire.cache.log-level=DEBUG
```

Or:

```
spring.data.gemfire.logging.level=DEBUG
```

If you previously used either of these SDG based logging properties, they will continue to work as designed in SBDG 1.3 or later.

16.1.2. Composing Logging Configuration

As mentioned earlier, SBDG allows you to compose your own logging configuration from SBDG's default, provided *Logback* configuration metadata.

SBDG conveniently bundles the Loggers and Appenders from SBDG's logging starter into a template file that you can include into your own, custom *Logback* XML configuration file.

The *Logback* template file appears as follows:

logback-include.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<included>

  <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d %5p %40.40c:%4L - %m%n</pattern>
    </encoder>
  </appender>

  <appender name="delegate"
class="org.springframework.geode.logging.slf4j.logback.DelegatingAppender"/>

  <logger name="com.gemstone.gemfire" level="${spring.boot.data.gemfire.log.level:-
INFO}" />
  <logger name="org.apache.geode" level="${spring.boot.data.gemfire.log.level:-
INFO}" />
  <logger name="org.jgroups" level="${spring.boot.data.gemfire.jgroups.log.level:-
ERROR}" />

</included>
```

Then, this Logback configuration snippet can be included in an application-specific, *Logback* XML configuration file as follows:

logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="false">

  <statusListener class="ch.qos.logback.core.status.NopStatusListener"/>

  <include resource="logback-include.xml"/>

  <root level="${logback.root.log.level:-INFO}">
    <appender-ref ref="console"/>
    <appender-ref ref="delegate"/>
  </root>

</configuration>
```

16.2. SLF4J & Logback API Support

SBDG provides additional support when working with the SLF4J and *Logback* APIs. This support is available when you declare the `org.springframework.geode:spring-geode-starter-logging` dependency on your Spring Boot application classpath.

One of the main supporting classes from the `spring-geode-starter-logger` is the

`org.springframework.geode.logging.slf4j.logback.LogbackSupport` class. This class provides methods to:

- Resolve a reference to the *Logback* `LoggingContext`
- Resolve the SLF4J ROOT `Logger` as a *Logback* `Logger`
- Lookup `Appenders` by name and required type
- Add/Remove `Appenders` to `Loggers`
- And even reset the state of the *Logback* logging system, which can prove to be most useful during testing

`LogbackSupport` can even suppress the auto-configuration of *Logback* performed by Spring Boot on startup, another useful utility during automated testing.

In addition to the `LogbackSupport` class, SBDG also provides some custom *Logback* `Appenders`.

16.2.1. CompositeAppender

The `org.springframework.geode.logging.slf4j.logback.CompositeAppender` class is an implementation of *Logback* `Appender` and the [Composite Software Design Pattern](#).

`CompositeAppender` enables developers to compose multiple `Appenders` and use them as if they were a single `Appender`.

For example, you could compose both the *Logback* `ConsoleAppender` and `FileAppender` into one using:

Composing multiple Appenders

```
class LoggingConfiguration {
    void composeAppenders() {

        ConsoleAppender<ILoggingEvent> consoleAppender = new ConsoleAppender<>();

        FileAppender<ILoggingEvent> fileAppender = new FileAppender<>();

        Appender<ILoggingEvent> compositeAppender =
        CompositeAppender.compose(consoleAppender, fileAppender);
    }
}

// do something with the compositeAppender
```

You could then add the `CompositeAppender` to a "named" `Logger` by doing:

Register `CompositeAppender` on "named" `Logger`

```
class LoggerConfiguration {
    void registerAppenderOnLogger() {

        Logger namedLogger = LoggerFactory.getLogger("loggerName");

        LogbackSupport.toLogbackLogger(namedLogger)
            .ifPresent(it -> LogbackSupport.addAppender(it, compositeAppender));
    }
}
```

In this case, the "named" `Logger` will log events (or log messages) to both the *Console* and *File* `Appenders`.

It is simple to compose an array or `Iterable` of `Appenders` by using either the `CompositeAppender.compose(:Appender<T>[])` method or the `CompositeAppender.compose(:Iterable<Appender<T>>)` method.

16.2.2. DelegatingAppender

The `org.springframework.geode.logging.slf4j.logback.DelegatingAppender` is a pass-through *Logback Appender* implementation wrapping another *Logback Appender*, or collection of `Appenders` doing actual work, like the `ConsoleAppender`, a `FileAppender` or a `SocketAppender`, etc. By default, the `DelegatingAppender` delegates to the `NOPAppender` thereby doing no actual work.

By default, SBDG registers the `org.springframework.geode.logging.slf4j.logback.DelegatingAppender` with the ROOT `Logger`, which can be useful for testing purposes.

With a reference to a `DelegatingAppender`, you can add any `Appender` as the delegate, even a `CompositeAppender`:

Add `ConsoleAppender` as the "delegate" for the `DelegatingAppender`

```
class LoggerConfiguration {
    void setupDelegation() {

        ConsoleAppender consoleAppender = new ConsoleAppender();

        LogbackSupport.resolveLoggerContext().ifPresent(consoleAppender::setContext);

        consoleAppender.setImmediateFlush(true);
        consoleAppender.start();

        LogbackSupport.resolveRootLogger()
            .flatMap(LogbackSupport::toLogbackLogger)
            .flatMap(rootLogger -> LogbackSupport.resolveAppender(rootLogger,
                LogbackSupport.DELEGATE_APPENDER_NAME, DelegatingAppender.class))
            .ifPresent(delegateAppender -> delegateAppender.setAppender(consoleAppender));
    }
}
```

16.2.3. StringAppender

The `org.springframework.geode.logging.slf4j.logback.StringAppender` stores log message in-memory, appended to a `String`.

The `StringAppender` is very useful for testing purposes. For instance, you can use the `StringAppender` to assert that a `Logger` used by certain application components logged messages at the appropriately configured log level while other log messages were not logged.

For example:

StringAppender in Action

```
class ApplicationComponent {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    public void someMethod() {
        logger.debug("Some debug message");
        // ...
    }

    public void someOtherMethod() {
        logger.info("Some info message");
    }
}

// Assuming the ApplicationComponent Logger was configured with log-level 'INFO',
// then...
class ApplicationComponentUnitTests {
```

```

    private final ApplicationComponent applicationComponent = new
ApplicationComponent();

    private final Logger logger = LoggerFactory.getLogger(ApplicationComponent.class);

    private StringAppender stringAppender;

    @Before
    public void setup() {

        LogbackSupport.toLogbackLogger(logger)
            .map(Logger::getLevel)
            .ifPresent(level -> assertThat(level).isEqualTo(Level.INFO));

        stringAppender = new StringAppender.Builder()
            .applyTo(logger)
            .build();
    }

    @Test
    public void someMethodDoesNotLogDebugMessage() {

        applicationComponent.someMethod();

        assertThat(stringAppender.getLogOutput()).doesNotContain("Some debug message");
    }

    @Test
    public void someOtherMethodLogsInfoMessage() {

        applicationComponent.someOtherMethod();

        assertThat(stringAppender.getLogOutput()).contains("Some info message");
    }
}

```

There are many other uses for the `StringAppender` and it can be used safely in a multi-Threaded context by calling `StringAppender.Builder.useSynchronization()`.

When combined with other SBDG provided `Appenders` in conjunction with the `LogbackSupport` class, you have a lot of power both in application code as well as your tests.

Chapter 17. Security

This section covers Security configuration for Apache Geode, which includes both Authentication & Authorization (collectively, Auth) as well as Transport Layer Security (TLS) using SSL.



Securing Data at Rest is not supported by Apache Geode.



Refer to the corresponding Sample [Guide](#) and [Code](#) to see Spring Boot Security for Apache Geode in action!

17.1. Authentication & Authorization

Apache Geode employs Username and Password based [Authentication](#) along with Role-based [Authorization](#) to secure your client to server data exchanges and operations.

Spring Data for Apache Geode provides [first-class support](#) for Apache Geode's Security framework, which is based on the [SecurityManager](#) interface. Additionally, Apache Geode's Security framework is integrated with [Apache Shiro](#), making the security for servers an even easier and more familiar task.



Eventually, support and integration with [Spring Security](#) will be provided by SBDG as well.

When you use Spring Boot for Apache Geode, which builds on the bits provided in Spring Data for Apache Geode, it makes short work of enabling Auth in both your clients and servers.

17.1.1. Auth for Servers

The easiest and most standard way to enable Auth in the servers of your cluster is to simply define 1 or more Apache Shiro [Realms](#) as beans in the Spring [ApplicationContext](#).

For example:

Declaring an Apache Shiro Realm

```
@Configuration
class ApacheGeodeSecurityConfiguration {

    @Bean
    DefaultLdapRealm ldapRealm() {
        return new DefaultLdapRealm();
    }

    // ...
}
```

When an Apache Shiro Realm (e.g. [DefaultLdapRealm](#)) is declared and registered in the Spring

`ApplicationContext` as a Spring bean, Spring Boot will automatically detect this `Realm` bean (or `Realm` beans if more than 1 is configured) and the Apache Geode servers in the cluster will automatically be configured with Authentication and Authorization enabled.

Alternatively, you can provide a custom, application-specific implementation of Apache Geode's `SecurityManager` interface, declared and registered as a bean in the Spring `ApplicationContext`:

Declaring a custom Apache Geode `SecurityManager`

```
@Configuration
class ApacheGeodeSecurityConfiguration {

    @Bean
    CustomSecurityManager customSecurityManager() {
        return new CustomSecurityManager();
    }

    // ...
}
```

Spring Boot will discover your custom, application-specific `SecurityManager` implementation and configure the servers in the Apache Geode cluster with Authentication and Authorization enabled.



The Spring team recommends that you use Apache Shiro to manage the Authentication & Authorization of your Apache Geode servers over implementing Apache Geode's `SecurityManager` interface.

17.1.2. Auth for Clients

When Apache Geode servers have been configured with Authentication & Authorization enabled, then clients must authenticate when connecting.

Spring Boot for Apache Geode makes this easy, regardless of whether you are running your Spring Boot, `ClientCache` applications in a local, non-managed environment or even when running in a cloud managed environment.

Non-Managed Auth for Clients

To enable Auth for clients connecting to a secure Apache Geode cluster, you simply only need to set a username and password in your Spring Boot `application.properties` file:

```
# Spring Boot client application.properties

spring.data.gemfire.security.username = jdoe
spring.data.gemfire.security.password = p@55w0rd
```

Spring Boot for Apache Geode will handle the rest.

Managed Auth for Clients

Enabling Auth for clients connecting to a Pivotal Cloud Cache (PCC) service instance in Pivotal CloudFoundry (PCF) is even easier.

You do not need to do anything!

When your Spring Boot application uses SBDG and is bound to PCC, then when you push (i.e. deploy) your app to PCF, Spring Boot for Apache Geode will extract the required Auth credentials from the environment that you setup when you provisioned a PCC service instance in your PCF organization & space. PCC automatically assigns 2 users with roles "*cluster_operator*" and "*developer*", respectively, to any Spring Boot application bound to the PCC service instance.

By default, SBDG will auto-configure your Spring Boot app to run with the user having the "*_cluster_operator*" Role. This ensures that your Spring Boot app has the necessary permissions (i.e. Authorization) to perform all data access operations on the servers in the PCC cluster including, for example, pushing configuration metadata from the client to the servers in the PCC cluster.

See the section, <<[cloudfoundry-cloudcache-security-auth-runtime-user-configuration,Running Spring Boot applications as a specific user]>>, in the [Pivotal Cloud Foundry](#) chapter for additional details on user authentication and authorization.

See the [chapter](#) titled '*Pivotal CloudFoundry*' for more general details.

See the [Pivotal Cloud Cache documentation](#) for security details when using PCC and PCF.

17.2. Transport Layer Security using SSL

Securing data in motion is also essential to the integrity of your application.

For instance, it would not do much good to send usernames and passwords over plain text Socket connections between your clients and servers, nor send sensitive data over those same connections.

Therefore, Apache Geode supports SSL between clients & servers, JMX clients (e.g. *Gfsh*) and the *Manager*, HTTP clients when using the Developer REST API or *Pulse*, between peers in the cluster, and when using the WAN Gateway to connect multiple sites (i.e. clusters).

Spring Data for Apache Geode provides [first-class support](#) for configuring and enabling SSL as well. Still, Spring Boot makes it even easier to configure and enable SSL, especially during development.

Apache Geode requires certain properties to be configured, which translate to the appropriate `javax.net.ssl.*` properties required by the JRE, to create Secure Socket Connections using [JSSE](#).

But, ensuring that you have set all the required SSL properties correctly is an error prone and tedious task. Therefore, Spring Boot for Apache Geode applies some basic conventions for you, out-of-the-box.

Simply create a `trusted.keystore`, JKS-based `KeyStore` file and place it in 1 of 3 well-known locations:

1. In your application JAR file at the root of the classpath.

2. In your Spring Boot application's working directory.
3. In your user home directory (as defined by the `user.home` Java System property).

When this file is named `trusted.keystore` and is placed in 1 of these 3 well-known locations, Spring Boot for Apache Geode will automatically configure your client to use SSL Socket connections.

If you are using Spring Boot to configure and bootstrap an Apache Geode server:

Spring Boot configured and bootstrapped Apache Geode server

```
@SpringBootApplication
@CacheServerApplication
class SpringBootApacheGeodeCacheServerApplication {
    // ...
}
```

Then, Spring Boot will apply the same procedure to enable SSL on the servers, between peers, as well.



During development it is convenient **not** to set a `trusted.keystore` password when accessing the keys in the JKS file. However, it is highly recommended that you secure the `trusted.keystore` file when deploying your application to a production environment.

If your `trusted.keystore` file is secured with a password, you will need to additionally specify the following property:

Accessing a secure `trusted.keystore`

```
# Spring Boot application.properties

spring.data.gemfire.security.ssl.keystore.password = p@55w0rd!
```

You can also configure the location of the keystore and truststore files, if they are separate, and have not been placed in 1 of the default, well-known locations searched by Spring Boot:

Accessing a secure `trusted.keystore`

```
# Spring Boot application.properties

spring.data.gemfire.security.ssl.keystore = /absolute/file/system/path/to/keystore.jks
spring.data.gemfire.security.ssl.keystore.password = keystorePassword
spring.data.gemfire.security.ssl.truststore =
/absolute/file/system/path/to/truststore.jks
spring.data.gemfire.security.ssl.truststore.password = truststorePassword
```

See the SDG [EnableSsl](#) annotation for all the configuration attributes and the corresponding properties expressed in `application.properties`.

17.3. Securing Data at Rest

Currently, neither Apache Geode nor Spring Boot or Spring Data for Apache Geode offer any support for securing your data while at rest (e.g. when your data has been overflowed or persisted to disk).

To secure data at rest when using Apache Geode, with or without Spring, you must employ 3rd party solutions like disk encryption, which is usually highly contextual and technology specific.

For example, to secure data at rest using Amazon EC2, see [Instance Store Encryption](#).

Chapter 18. Testing

Spring Boot for Apache Geode (SBDG), with help from [Spring Test for Apache Geode \(STDG\)](#), offers first-class support for both *Unit* & *Integration Testing* of Apache Geode in your Spring Boot applications.



See the Spring Test for Apache Geode (STDG) [documentation](#) for more details.

18.1. Unit Testing

Unit Testing with Apache Geode using *mock objects* in a Spring Boot Test is as simple as declaring the STDG `@EnableGemFireMockObjects` annotation in your test configuration:


```

@SpringBootTest
@RunWith(SpringRunner.class)
public class SpringBootApacheGeodeUnitTest extends IntegrationTestsSupport {

    @Autowired
    private UserRepository userRepository;

    @Test
    public void saveAndFindUserIsSuccessful() {

        User jonDoe = User.as("jonDoe");

        assertThat(this.userRepository.save(jonDoe)).isNotNull();

        User jonDoeFoundById =
this.userRepository.findById(jonDoe.getName()).orElse(null);

        assertThat(jonDoeFoundById).isEqualTo(jonDoe);
    }

    @SpringBootApplication
    @EnableGemFireMockObjects
    @EnableEntityDefinedRegions(basePackageClasses = User.class)
    static class TestConfiguration { }

}

@Getter
@ToString
@EqualsAndHashCode
@RequiredArgsConstructor(staticName = "as")
@Region("Users")
class User {

    @Id
    @lombok.NonNull
    private String name;

}

interface UserRepository extends CrudRepository<User, String> { }

```

While this test class is not a "pure" *Unit Test*, particularly since it bootstraps an actual Spring `ApplicationContext` using Spring Boot, it does, however, "mock" all Apache Geode objects, such as the "Users" `Region` declared by the `User` application entity class, which was annotated with SDG's `@Region` mapping annotation.

This test class conveniently uses Spring Boot's *auto-configuration* to auto-configure an Apache

Geode `ClientCache` instance. In addition, SDG's `@EnableEntityDefinedRegions` annotation was used to conveniently create the Apache Geode "Users" `Region` to store instances of `User`.

Finally, Spring Data's `Repository` abstraction was used to conveniently perform basic CRUD (e.g. `save`) and simple (OQL) query (e.g. `findById`) data access operations on the "Users" `Region`.

Even though the Apache Geode objects (e.g. "Users" `Region`) are "mock objects", you can still perform many of the data access operations required by your Spring Boot application's business logic in a Apache Geode API agnostic way, that is, using Spring's powerful programming model and constructs!



By extending STDG's `org.springframework.data.gemfire.tests.integration.IntegrationTestSupport` class, you ensure that all Apache Geode mock objects and resources are properly released after the test class runs, thereby preventing any interference with downstream tests.

While STDG tries to `mock the functionality and behavior` for many `Region` operations, it is simply not pragmatic to mock them all. For example, it would not be practical to mock `Region` query operations involving complex OQL statements having sophisticated predicates.

If such functional testing is required, then the test might be better suited as an *Integration Test*. Alternatively, you can follow the advice in this [section](#).

In general, STDG provides the following capabilities when mocking Apache Geode objects out-of-the-box:

- [Mock Object Scope & Lifecycle Management](#)
- [Support for Mock Regions with Data](#)
- [Support for Mocking Region Callbacks](#)
- [Support for Mocking Unsupported Region Operations](#)



See documentation on [Unit Testing with STDG](#) for more details.

18.2. Integration Testing

Integration Testing with Apache Geode in a Spring Boot Test is as simple as **not** declaring STDG's `@EnableGemFireMockObjects` annotation in your test configuration. Of course, you may then want to additionally use SBDG's `@EnableClusterAware` annotation to conditionally detect the presence of a Apache Geode cluster:

Using `@EnableClusterAware` in test configuration

```
@SpringBootApplication
@EnableClusterAware
@EnableEntityDefinedRegions(basePackageClasses = User.class)
static class TestConfiguration { }
```

The SBDG `@EnableClusterAware` annotation will conveniently toggle your auto-configured `ClientCache` instance between local-only mode and client/server. Additionally, it will even push configuration metadata (e.g. `Region` definitions) up to the server(s) in the cluster required by the application to persist data.

In most cases, in addition to testing with "live" Apache Geode objects (e.g. *Regions*), we also want to test in a client/server capacity. This unlocks the full capabilities of the Apache Geode data management system in a Spring context, and gets you as close as possible to production from the comfort of your IDE.

Building on our example from the section on [Unit Testing](#), you can modify the test to use "live" Apache Geode objects in a client/server topology as follows:

Integration Test with Apache Geode using Spring Boot

```
@ActiveProfiles("client")
@RunWith(SpringRunner.class)
@SpringBootTest(properties = "spring.data.gemfire.management.use-http=false")
public class SpringBootApacheGeodeIntegrationTest extends
    ForkingClientServerIntegrationTestsSupport {

    @BeforeClass
    public static void startGeodeServer() throws IOException {
        startGemFireServer(TestGeodeServerConfiguration.class);
    }

    @Autowired
    private UserRepository userRepository;

    @Test
    public void saveAndFindUserIsSuccessful() {

        User jonDoe = User.as("jonDoe");

        assertThat(this.userRepository.save(jonDoe)).isNotNull();

        User jonDoeFoundById =
            this.userRepository.findById(jonDoe.getName()).orElse(null);

        assertThat(jonDoeFoundById).isEqualTo(jonDoe);
        assertThat(jonDoeFoundById).isNotSameAs(jonDoe);
    }

    @SpringBootApplication
    @EnableClusterAware
    @EnableEntityDefinedRegions(basePackageClasses = User.class)
    @Profile("client")
    static class TestGeodeClientConfiguration { }

    @CacheServerApplication
    @Profile("server")
```

```

static class TestGeodeServerConfiguration {

    public static void main(String[] args) {

        new SpringApplicationBuilder(TestGeodeServerConfiguration.class)
            .web(WebApplicationType.NONE)
            .profiles("server")
            .build()
            .run(args);
    }
}

@Getter
@ToString
@EqualsAndHashCode
@RequiredArgsConstructor(staticName = "as")
@Region("Users")
class User {

    @Id
    @lombok.NonNull
    private String name;
}

interface UserRepository extends CrudRepository<User, String> { }

```

The application client/server-based *Integration Test* class extend STDG's `org.springframework.data.gemfire.tests.integration.ForkingClientServerIntegrationTestsSupport` class. This ensures that all Apache Geode objects and resources are properly cleaned up after the test class runs. In addition, it coordinates the client & server components of the test (e.g. connecting the client to the server using a random port).

The server is started in a `@BeforeClass` setup method:

Start the Apache Geode server

```

class SpringBootApacheGeodeIntegrationTest extends
ForkingClientServerIntegrationTestsSupport {

    @BeforeClass
    public static void startGeodeServer() throws IOException {
        startGemFireServer(TestGeodeServerConfiguration.class);
    }
}

```

STDG allows you to configure the server with Spring config, specified in the `TestGeodeServerConfiguration` class. The Java class needs to provide a `main` method. It uses the `SpringApplicationBuilder` to bootstrap the Apache Geode `CacheServer` application.

```
@CacheServerApplication
@Profile("server")
static class TestGeodeServerConfiguration {

    public static void main(String[] args) {

        new SpringApplicationBuilder(TestGeodeServerConfiguration.class)
            .web(WebApplicationType.NONE)
            .profiles("server")
            .build()
            .run(args);
    }
}
```

In this case, we provide very minimal configuration since the configuration is determined and pushed up to the server by the client. For example, we do not need to explicitly create the "Users" **Region** on the server-side since it is implicitly handled for you by the SBDG/STDG frameworks from the client.

We take advantage of Spring *Profiles* in the test setup to distinguish between the client & server configuration. Keep in mind that the test is the "client" in this arrangement.

The STDG framework is doing as the supporting class states, "forking" the Spring Boot-based, Apache Geode **CacheServer** application in a separate JVM process. Subsequently, the STDG framework will stop the server upon completion of the tests in the test class.

Of course, you are free to start your server(s) or cluster however you choose. STDG simply and conveniently provides this capability for you since it is a common concern.

This test class is very simple and much more complex test scenarios can be easily handled by STDG.



Review SBDG's test suite to witness the full power and functionality of the STDG framework for yourself.



See documentation on [Integration Testing with STDG](#) for more details.

Chapter 19. Apache Geode API Extensions

The Spring Boot for Apache Geode (SBDG) project includes the `org.springframework.geode:apache-geode-extensions` module to make using Apache Geode APIs tolerable and useful. While this module is relatively new, it contains several API extensions already.

Apache Geode's API is quite convoluted with many design problems:

1. Non-intuitive, complex interfaces that contradict industry standard terms. (e.g. `Cache` vs. `Region`).
2. APIs with an excessive footprint and no sensible ADTs resulting in too many overloaded methods with loaded method signatures (e.g. `Region`).
3. Lingering deprecations causing excess baggage.
4. Use of public fields exposing internal state, violating encapsulation, making it difficult to uphold invariants.
5. Useful functionality hidden behind so called "internal" APIs that should be public.
6. Utility/Helper classes containing functionality that should be part of the types on which the Utility class operates (e.g. `PartitionRegionHelper`).
7. Incorrect use of *Checked Exceptions* (e.g. `IncompatibleVersionException`).
8. Inconsistent behavior across different methods of configuration: API vs. `cache.xml` vs. `Cluster Configuration` using `Gfsh`.
9. APIs closed for modification, yet offer no option for extension thereby violating the *Open/Closed Principle*.
10. In general, poor *Separation of Concerns* (e.g. `Region`), violating many of the **SOLID** principles.
11. Components (e.g. `Pool`) that are difficult to test properly: Apache Geode often incorrectly refers to implementation classes rather than interfaces leading to `ClassCastExceptions` and violation of the *Program to Interfaces* principle.
12. Excessive use of `static` initializer blocks making Apache Geode difficult to test.
13. Untimely shutdown and release of resources that run interference when writing *Integration Tests*.

This list goes on making Apache Geode's APIs difficult and confusing to use at times, especially without prior knowledge or experience. Users very often get this wrong and it is the main reason why Spring's APIs for Apache Geode are so invaluable; they can help you do the right thing!

Let's consider a few examples.

The one and only cache implementation (`GemFireCacheImpl`) implements both the `ClientCache` and `Cache` interfaces. A `ClientCache` instance is created by client applications to access and persist data in a Apache Geode cluster. On the contrary, a *peer* `Cache` instance is created by server-side applications serving as peer members of the Apache Geode cluster and distributed system to manage data. Both incarnations result in an instance of `GemFireCacheImpl`, yet a cache cannot be both a client and a peer. But, you would never know this by introspecting the cache instance.

The `Delta` interface, `hasDelta()` method, is another point of confusion. If there is no delta, why send

the object in its entirety? Presumably there are no changes. Of course, there is a reason but it is not immediately apparent why given the lack of documentation.

Spring in general, and SBDG in particular, shield users from design problems as well as changes to Apache Geode's APIs that could adversely affect your applications when integrating with Apache Geode. Spring's APIs provide a layer of indirection along with enhanced capabilities (e.g. Exception translation).



Spring Data for Apache Geode (SDG) also [offers](#) some relief when using Apache Geode's APIs.

19.1. SimpleCacheResolver

In some cases, it is necessary to acquire a reference to the cache instance in your application components at runtime. For example, you might want to create a temporary [Region](#) on the fly in order to aggregate data for analysis.

Typically, you already know the type of cache your application is using since you must declare your application to be either a client (i.e. [ClientCache](#)) in the [client/server topology](#), or a [peer member/node](#) in the cluster (i.e. [Cache](#)) on startup. This is expressed in configuration when creating the cache instance required to interact with the Apache Geode data management system. In most cases, your application will be a client and SBDG makes this decision easy since it *auto-configures* a [ClientCache](#) instance, [by default](#).

In a Spring context, the cache instance created by the framework is a managed bean in the Spring container. As such, it is a simple matter to inject a reference to the *Singleton* cache bean into any other managed application component.

Autowired Cache Reference using Dependency Injection (DI)

```
@Service
class CacheMonitoringService {

    @Autowired
    ClientCache clientCache;

    // use the clientCache object reference to monitor the cache as necessary

}
```

However, in cases where your application component or class is not managed by Spring and you need a reference to the cache instance at runtime, SBDG provides the abstract [org.springframework.geode.cache.SimpleCacheResolver](#) class (see [Javadoc](#)).

```
package org.springframework.geode.cache;

abstract class SimpleCacheResolver {

    <T extends GemFireCache> T require() { }

    <T extends GemFireCache> Optional<T> resolve() { }

    Optional<ClientCache> resolveClientCache() { }

    Optional<Cache> resolvePeerCache() { }

}
```

`SimpleCacheResolver` adheres to [SOLID OO Principles](#). This class is abstract and extensible so users can change the algorithm used to resolve client or peer cache instances as well as mock its methods in *Unit Tests*.

Additionally, each method is precise. For example, `resolveClientCache()` will only resolve a reference to a cache if the cache instance is a "client"! If a cache exists, but is a "peer" instance, then `resolveClientCache()` returns `Optional.EMPTY`. The behavior of `resolvePeerCache()` is similar.

`require()` returns a non-`Optional` reference to a cache instance throwing an `IllegalStateException` if a cache is not present.

19.2. CacheUtils

Under-the-hood, `SimpleCacheResolver` delegates some of its functions to the `CacheUtils` abstract utility class, which provides additional, convenient capabilities when using a cache.

While there are utility methods to determine whether a cache instance (i.e. `GemFireCache`) or *Region* is a client or a peer, one of the more useful functions is to extract all the values from a *Region*.

To extract all the values stored in a *Region* call `CacheUtils.collectValues(:Region<?, T>)`. This method returns a `Collection<T>` containing all the values stored in the given *Region*. The method is smart, and knows how to handle the *Region* appropriately regardless of whether the *Region* is a client or peer *Region*. This distinction is important since client *PROXY Regions* store no values.



Caution is advised when getting all values from a *Region*. While getting filtered reference values from a non-transactional, reference data only `[REPLICATE]` *Region* is quite useful, getting all values from a transactional, `[PARTITION]` *Region* can prove quite detrimental, especially in production. Getting all values from a *Region* can be useful during testing.

19.3. `MembershipListenerAdapter` & `MembershipEvent`

Another useful API hidden by Apache Geode is the membership events and listener interface. This API is especially useful on the server-side when your Spring Boot application is serving as a peer member of an Apache Geode distributed system.

When a peer member is disconnected from the distributed system, perhaps due to a network failure, the member is forcibly removed from the cluster. This node immediately enters a reconnecting state, trying to establish a connection back to the cluster. Once reconnected, the peer member must rebuild all cache objects (i.e. `Cache`, `Regions`, `Indexes`, `DiskStores`, etc). All previous cache objects are now invalid and their references stale.

As you can imagine, in a Spring context this is particularly problematic since most Apache Geode objects are *Singleton* beans declared in and managed by the Spring container. Those beans may be injected and used in other framework and application components. For instance, `Regions` are injected into SDG's `GemfireTemplate`, Spring Data *Repositories* and possibly application-specific *Data Access Objects* (DAO).

If references to those cache objects become stale on a forced disconnect event, then there is no way to auto-wire fresh object references into the dependent application or framework components when the peer member is reconnected unless the Spring `ApplicationContext` is "refreshed". In fact, there is no way to even know that this event has occurred since the Apache Geode `MembershipListener` API and corresponding events are "internal".



The Spring team have explored the idea of creating proxies for all types of cache objects (i.e. `Cache`, `Regions`, `Indexes`, `DiskStores`, `AsyncEventQueues`, `GatewayReceivers`, `GatewaySenders`, etc) used by Spring. The proxies would know how to obtain a "fresh" reference on a reconnect event. However, this turns out to be more problematic than it is worth. It is simply easier to "refresh" the Spring `ApplicationContext`, although no less cheap. Neither way is ideal. See [SGF-921](#) and [SGF-227](#) for further details.

In the case where membership events are useful to the Spring Boot application, SBDG provides the following [API](#):

- `MembershipListenerAdapter`
- `MembershipEvent`

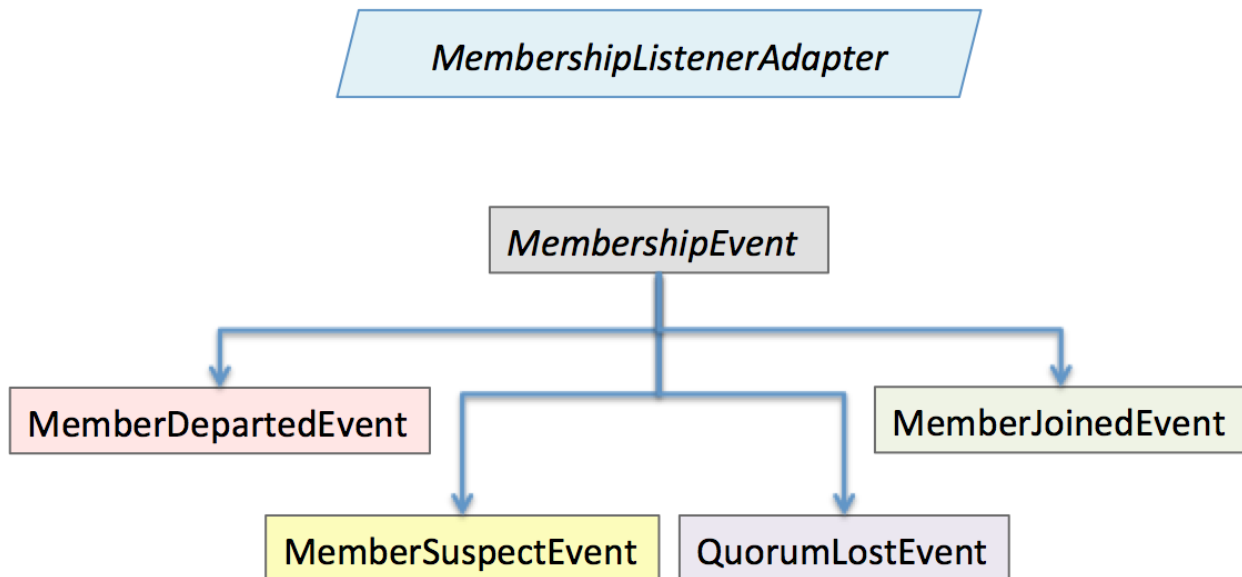
The abstract `MembershipListenerAdapter` class implements Apache Geode's clumsy `org.apache.geode.distributed.internal.MembershipListener` interface to simplify the event handler method signatures by using an appropriate `MembershipEvent` type to encapsulate the actors in the event.

The abstract `MembershipEvent` class is further subclassed to represent specific membership event types that occur within the Apache Geode system:

- `MemberDepartedEvent`
- `MemberJoinedEvent`

- `MemberSuspectEvent`
- `QuorumLostEvent`

The API is depicted in this UML diagram:



The membership event type is further categorized with an appropriate enumerated value, `MembershipEvent.Type`, as a property of the `MembershipEvent` itself (see `getType()`).

The type hierarchy is useful in `instanceof` expressions while the `Enum` is useful in `switch` statements.

You can see 1 particular implementation of the `MembershipListenerAdapter` with the `ApplicationContextMembershipListener` class, which does exactly as we described above, handling forced-disconnect/auto-reconnect membership events inside a Spring context in order to refresh the Spring `ApplicationContext`.

19.4. PDX

Apache Geode's PDX serialization framework is yet another API that falls short of a complete stack.

For instance, there is no easy or direct way to serialize an object as PDX bytes. It is also not possible to modify an existing `PdxInstance` by adding or removing fields since it requires a new PDX type. In this case, you must create a new `PdxInstance` and copy from the existing `PdxInstance`. Unfortunately, the Apache Geode API offers no assistance. It is also not possible to use PDX in a client, local-only mode without a server since the PDX type registry is only available and managed on servers in a cluster. All of this leaves much to be desired.

19.4.1. `PdxInstanceBuilder`

In such cases, SBDG conveniently provides the `PdxInstanceBuilder` class, appropriately named after the *Builder Software Design Pattern*. The `PdxInstanceBuilder` also offers a fluent API for constructing

PdxInstances.

PdxInstanceBuilder *API*

```
class PdxInstanceBuilder {  
    PdxInstanceFactory copy(PdxInstance pdx);  
    Factory from(Object target);  
}
```

For example, you could serialize an application domain object as PDX bytes with the following code:

Serializing an Object to PDX

```
@Component  
class CustomerSerializer {  
    PdxInstance serialize(Customer customer) {  
        return PdxInstanceBuilder.create()  
            .from(customer)  
            .create();  
    }  
}
```

You could then modify the **PdxInstance** by copying from the original:

Copy PdxInstance

```
@Component  
class CustomerDecorator {  
    @Autowired  
    CustomerSerializer serializer;  
    PdxIntance decorate(Customer customer) {  
        PdxInstance pdxCustomer = serializer.serialize(customer);  
        return PdxInstanceBuilder.create()  
            .copy(pdxCustomer)  
            .writeBoolean("vip", isImportant(customer))  
            .create();  
    }  
}
```

19.4.2. PdxInstanceWrapper

SBDG also provides the `PdxInstanceWrapper` class to wrap an existing `PdxInstance` in order to provide more control during the conversion from PDX to JSON and from JSON back into a POJO. Specifically, the wrapper gives users more control over the configuration of Jackson's `ObjectMapper`.

The `ObjectMapper` constructed by Apache Geode's own `PdxInstance` implementation (`PdxInstanceImpl`) is not configurable nor was it configured correctly. And unfortunately, since `PdxInstance` is not extensible, the `getObject()` method fails miserably when converting the JSON generated from PDX back into a POJO for any practical application domain model type.

Wrapping an existing `PdxInstance`

```
PdxInstanceWrapper wrapper = PdxInstanceWrapper.from(pdxInstance);
```

For all operations on `PdxInstance` except `getObject()`, the wrapper delegates to the underlying `PdxInstance` method implementation called by the user.

In addition to the decorated `getObject()` method, the `PdxInstanceWrapper` provides a thorough implementation of the `toString()` method. The state of the `PdxInstance` is output in a JSON-like String.

Finally, the `PdxInstanceWrapper` class adds a `getIdentifier()` method. Rather than put the burden on the user to have to iterate the field names of the `PdxInstance` to determine whether a field is the identity field, and then call `getField(..)` with the field name to get the ID (value), assuming an identity field was marked in the first place, the `PdxInstanceWrapper` class provides the `getIdentifier()` method to return the ID of the `PdxInstance` directly.

The `getIdentifier()` method is smart in that it first iterates the fields of the `PdxInstance` asking if the field is the identity field. If no field was marked as the "identity" field, then the algorithm searches for a field named "id". If no field with the name "id" exists, then the algorithm searches for a metadata field called "@identifier", which refers to the field that is the identity field of the `PdxInstance`.

The `@identifier` metadata field is useful in cases where the `PdxInstance` originated from JSON and the application domain object uses a natural identifier, rather than a surrogate ID, such as `Book.isbn`.



Apache Geode's `JSONFormatter` is not capable of marking the identity field of a `PdxInstance` originating from JSON.



It is not currently possible to implement the `PdxInstance` interface and store instances of this type as a value in a `Region`. Apache Geode naively assumes that all `PdxInstance` objects are an implementation created by Apache Geode itself (i.e. `PdxInstanceImpl`), which has a tight coupling to the PDX type registry. An Exception is thrown if you try to store instances of your own `PdxInstance` implementation.

19.4.3. ObjectPdxInstanceAdapter

In rare cases, it might be necessary to treat an `Object` as a `PdxInstance` depending on the context without incurring the overhead of serializing an `Object` to PDX. For such cases, SBDG offers the `ObjectPdxInstanceAdapter` class.

This might be true when calling a method with a parameter expecting an argument, or returning an instance, of type `PdxInstance`, particularly when Apache Geode's `read-serialized` PDX configuration property is set to `true`, and only an object is available in the current context.

Under-the-hood, SBDG's `ObjectPdxInstanceAdapter` class uses Spring's `BeanWrapper` class along with *Java's Introspection & Reflection* functionality to adapt the given `Object` in order to access it using the full `PdxInstance` API. This includes the use of the `WritablePdxInstance` API, obtained from `PdxInstance.createWriter()`, to modify the underlying `Object` as well.

Like the `PdxInstanceWrapper` class, `ObjectPdxInstanceAdapter` contains special logic to resolve the identity field and ID of the `PdxInstance`, including consideration for Spring Data's `@Id` mapping annotation, which can be introspected in this case given the underlying `Object` backing the `PdxInstance` is a POJO.

Clearly, the `ObjectPdxInstanceAdapter.getObject()` method will return the given, wrapped `Object` used to construct the `ObjectPdxInstanceAdapter`, and is therefore, automatically "*deserializable*", as determined by the `PdxInstance.isDeserializable()` method, which always returns true.

To adapt any `Object` as a `PdxInstance`, simply do:

Adapt an `Object` as a `PdxInstance`

```
class OfflineObjectToPdxInstanceConverter {  
  
    @NonNull PdxInstance convert(@NonNull Object target) {  
        return ObjectPdxInstanceAdapter.from(target);  
    }  
}
```

Once the adapter is created, you can use it to access data on the underlying `Object`.

For example, given a `Customer` class:

Customer class

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

    @Id
    private Long id;

    String name;

    // constructors, getters and setters omitted

}
```

Then accessing an instance of *Customer* using the *PdxInstance* API is as easy as:

Accessing an Object using the PdxInstance API

```
class ObjectPdxInstanceAdapterTest {

    @Test
    public void getAndSetObjectProperties() {

        Customer jonDoe = new Customer(1L, "Jon Doe");

        PdxInstance adapter = ObjectPdxInstanceAdapter.from(jonDoe);

        assertThat(jonDoe.getName()).isEqualTo("Jon Doe");
        assertThat(adapter.getField("name")).isEqualTo("Jon Doe");

        adapter.createWriter().setField("name", "Jane Doe");

        assertThat(adapter.getField("name")).isEqualTo("Jane Doe");
        assertThat(jonDoe.getName()).isEqualTo("Jane Doe");
    }

}
```

19.5. Security

For testing purposes, SBDG provides a test implementation of Apache Geode's *SecurityManager* interface that simply expects the password to match the username (case-sensitive) when authenticating.

By default, all operations are authorized.

To match the expectations of SBDG's *TestSecurityManager*, SBDG additionally provides a test implementation of Apache Geode's *AuthInitialize* interface that supplies matching credentials for both the username and password.

Chapter 20. Spring Boot Actuator

Spring Boot for Apache Geode and VMware Tanzu GemFire (SBDG) adds [Spring Boot Actuator](#) support and dedicated [HealthIndicators](#) for Apache Geode and VMware Tanzu GemFire. Equally, the provided [HealthIndicators](#) will even work with Pivotal Cloud Cache, which is backed by VMware Tanzu GemFire, when pushing your Spring Boot applications to Pivotal CloudFoundry (PCC).

Spring Boot [HealthIndicators](#) provide details about the runtime operation and behavior of your Apache Geode based Spring Boot applications. For instance, by querying the right [HealthIndicator](#) endpoint, you would be able to get the current hit/miss count for your [Region.get\(key\)](#) data access operations.

In addition to vital health information, SBDG provides basic, pre-runtime configuration meta-data about the Apache Geode components that are monitored by Spring Boot Actuator. This makes it easier to see how the application was configured all in one place, rather than in properties files, Spring config, XML, etc.

The provided Spring Boot [HealthIndicators](#) fall under one of three categories:

- Base [HealthIndicators](#) that apply to all Apache Geode, Spring Boot applications, regardless of cache type, such as Regions, Indexes and DiskStores.
- Peer [Cache](#) based [HealthIndicators](#) that are only applicable to peer [Cache](#) applications, such as [AsyncEventQueues](#), [CacheServers](#), [GatewayReceivers](#) and [GatewaySenders](#).
- And finally, [ClientCache](#) based [HealthIndicators](#) that are only applicable to [ClientCache](#) applications, such as [ContinuousQueries](#) and connection [Pools](#).

The following sections give a brief overview of all the available Spring Boot [HealthIndicators](#) provided for Apache Geode out-of-the-box.



Refer to the corresponding Sample [Guide](#) and [Code](#) to see the Spring Boot Actuator for Apache Geode in action!

20.1. Base [HealthIndicators](#)

The following section covers Spring Boot [HealthIndicators](#) that apply to both peer [Cache](#) and [ClientCache](#), Spring Boot applications. That is, these [HealthIndicators](#) are not specific to the cache type.

In Apache Geode, the cache instance is either a peer [Cache](#) instance, which makes your Spring Boot application part of a Apache Geode cluster, or more commonly, a [ClientCache](#) instance that talks to an existing cluster. Your Spring Boot application can only be one cache type or the other and can only have a single instance of that cache type.

20.1.1. GeodeCacheHealthIndicator

The [GeodeCacheHealthIndicator](#) provides essential details about the (single) cache instance (Client or

Peer) along with the underlying `DistributedSystem`, the `DistributedMember` and configuration details of the `ResourceManager`.

When your Spring Boot application creates an instance of a peer `Cache`, the `DistributedMember` object represents your application as a peer member/node of the `DistributedSystem` formed from a collection of connected peers (i.e. the cluster), to which your application also has `access`, indirectly via the cache instance.

This is no different for a `ClientCache` even though the client is technically not part of the peer/server cluster. But, it still creates instances of the `DistributedSystem` and `DistributedMember` objects, respectively.

The following configuration meta-data and health details about each object is covered:

Table 1. Cache Details

Name	Description
geode.cache.name	Name of the member in the distributed system.
geode.cache.closed	Determines whether the cache has been closed.
geode.cache.cancel-in-progress	Cancellation of operations in progress.

Table 2. DistributedMember Details

Name	Description
geode.distributed-member.id	DistributedMember identifier (used in logs internally).
geode.distributed-member.name	Name of the member in the distributed system.
geode.distributed-members.groups	Configured groups to which the member belongs.
geode.distributed-members.host	Name of the machine on which the member is running.
geode.distributed-members.process-id	Identifier of the JVM process (PID).

Table 3. DistributedSystem Details

Name	Description
geode.distributed-system.member-count	Total number of members in the cluster (1 for clients).
geode.distributed-system.connected	Indicates whether the member is currently connected to the cluster.

Name	Description
geode.distributed-system.reconnecting	Indicates whether the member is in a reconnecting state, which happens when a network partition occurs and the member gets disconnected from the cluster.
geode.distributed-system.properties-location	Location of the standard configuration properties .
geode.distributed-system.security-properties-location	Location of the security configuration properties .

Table 4. ResourceManager Details

Name	Description
geode.resource-manager.critical-heap-percentage	Percentage of heap at which the cache is in danger of becoming inoperable.
geode.resource-manager.critical-off-heap-percentage	Percentage of off-heap at which the cache is in danger of becoming inoperable.
geode.resource-manager.eviction-heap-percentage	Percentage of heap at which eviction begins on Regions configured with a Heap LRU Eviction policy.
geode.resource-manager.eviction-off-heap-percentage	Percentage of off-heap at which eviction begins on Regions configured with a Heap LRU Eviction policy.

20.1.2. GeodeRegionsHealthIndicator

The **GeodeRegionsHealthIndicator** provides details about all the configured and known **Regions** in the cache. If the cache is a client, then details will include all LOCAL, PROXY and CACHING_PROXY **Regions**. If the cache is a peer, then the details will include all LOCAL, PARTITION and REPLICATE **Regions**.

While the configuration meta-data details are not exhaustive, essential details along with basic performance metrics are covered:

Table 5. Region Details

Name	Description
geode.cache.region.s.<name>.cloning-enabled	Whether Region values are cloned on read (e.g. cloning-enabled is true when cache transactions are used to prevent in-place modifications).

Name	Description
geode.cache.region s.<name>.data-policy	Policy used to manage the data in the Region (e.g. PARTITION, REPLICATE, etc).
geode.cache.region s.<name>.initial-capacity	Initial number of entries that can be held by a Region before it needs to be resized.
geode.cache.region s.<name>.load-factor	Load factor used to determine when to resize the Region when it nears capacity.
geode.cache.region s.<name>.key-constraint	Type constraint for Region keys.
geode.cache.region s.<name>.off-heap	Determines whether this Region will store values in off-heap memory (NOTE: Keys are always kept on Heap).
geode.cache.region s.<name>.pool-name	If this Region is a client Region, then this property determines the configured connection Pool (NOTE: Regions can have and use dedicated Pools for their data access operations.)
geode.cache.region s.<name>.pool-name	Determines the Scope of the Region, which plays a factor in the Regions consistency-level, as it pertains to acknowledgements for writes.
geode.cache.region s.<name>.value-constraint	Type constraint for Region values.

Additionally, when the Region is a peer **Cache PARTITION** Region, then the following details are also covered:

Table 6. Partition Region Details

Name	Description
geode.cache.region s.<name>.partition. collocated-with	Indicates this Region is collocated with another PARTITION Region, which is necessary when performing equi-joins queries (NOTE: distributed joins are not supported).
geode.cache.region s.<name>.partition. local-max-memory	Total amount of Heap memory allowed to be used by this Region on this node.
geode.cache.region s.<name>.partition. redundant-copies	Number of replicas for this PARTITION Region, which is useful in High Availability (HA) use cases.
geode.cache.region s.<name>.partition. total-max-memory	Total amount of Heap memory allowed to be used by this Region across all nodes in the cluster hosting this Region.

Name	Description
geode.cache.region s.<name>.partition. total-number-of- buckets	Total number of buckets (shards) that this Region is divided up into (NOTE: defaults to 113).

Finally, when statistics are enabled (e.g. using `@EnableStatistics`, (see [here](#) for more details), the following details are available:

Table 7. Region Statistic Details

Name	Description
geode.cache.region s.<name>.statistics. hit-count	Number of hits for a Region entry.
geode.cache.region s.<name>.statistics. hit-ratio	Ratio of hits to the number of <code>Region.get(key)</code> calls.
geode.cache.region s.<name>.statistics. last-accessed-time	For an entry, determines the last time it was accessed with <code>Region.get(key)</code> .
geode.cache.region s.<name>.statistics. last-modified-time	For an entry, determines the time a Region's entry value was last modified.
geode.cache.region s.<name>.statistics. miss-count	Returns the number of times that a <code>Region.get</code> was performed and no value was found locally.

20.1.3. GeodeIndexesHealthIndicator

The `GeodeIndexesHealthIndicator` provides details about the configured Region `Indexes` used in OQL query data access operations.

The following details are covered:

Table 8. Index Details

Name	Description
geode.index.<name>. >.from-clause	Region from which data is selected.
geode.index.<name>. >.indexed- expression	The Region value fields/properties used in the Index expression.
geode.index.<name>. >.projection- attributes	For all other Indexes, returns "", but for Map Indexes, returns either "" or the specific Map keys that were indexed.

Name	Description
geode.index.<name>.region	Region to which the Index is applied.

Additionally, when statistics are enabled (e.g. using `@EnableStatistics`; (see [here](#) for more details), the following details are available:

Table 9. Index Statistic Details

Name	Description
geode.index.<name>.statistics.number-of-bucket-indexes	Number of bucket Indexes created in a Partitioned Region.
geode.index.<name>.statistics.number-of-keys	Number of keys in this Index.
geode.index.<name>.statistics.number-of-map-indexed-keys	Number of keys in this Index at the highest-level.
geode.index.<name>.statistics.number-of-values	Number of values in this Index.
geode.index.<name>.statistics.number-of-updates	Number of times this Index has been updated.
geode.index.<name>.statistics.read-lock-count	Number of read locks taken on this Index.
geode.index.<name>.statistics.total-update-time	Total amount of time (ns) spent updating this Index.
geode.index.<name>.statistics.total-uses	Total number of times this Index has been accessed by an OQL query.

20.1.4. GeodeDiskStoresHealthIndicator

The `GeodeDiskStoresHealthIndicator` provides details about the configured `DiskStores` in the system/application. Remember, `DiskStores` are used to overflow and persist data to disk, including type meta-data tracked by PDX when the values in the Region(s) have been serialized with PDX and the Region(s) are persistent.

Most of the tracked health information pertains to configuration:

Table 10. DiskStore Details

Name	Description
geode.disk-store.<name>.allow-force-compaction	Indicates whether manual compaction of the DiskStore is allowed.
geode.disk-store.<name>.auto-compact	Indicates if compaction occurs automatically.
geode.disk-store.<name>.compaction-threshold	Percentage at which the oplog will become compactable.
geode.disk-store.<name>.disk-directories	Location of the oplog disk files.
geode.disk-store.<name>.disk-directory-sizes	Configured and allowed sizes (MB) for the disk directory storing the disk files.
geode.disk-store.<name>.disk-usage-critical-percentage	Critical threshold of disk usage proportional to the total disk volume.
geode.disk-store.<name>.disk-usage-warning-percentage	Warning threshold of disk usage proportional to the total disk volume.
geode.disk-store.<name>.max-oplog-size	Maximum size (MB) allowed for a single oplog file.
geode.disk-store.<name>.queue-size	Size of the queue used to batch writes flushed to disk.
geode.disk-store.<name>.time-interval	Time to wait (ms) before writes are flushed to disk from the queue if the size limit has not be reached.
geode.disk-store.<name>.uuid	Universally Unique Identifier for the DiskStore across Distributed System.
geode.disk-store.<name>.write-buffer-size	Size the of write buffer the DiskStore uses to write data to disk.

20.2. ClientCache HealthIndicators

The **ClientCache** based **HealthIndicators** provide additional details specifically for Spring Boot, cache client applications. These **HealthIndicators** are only available when the Spring Boot application creates a **ClientCache** instance (i.e. is a cache client), which is the default.

20.2.1. GeodeContinuousQueriesHealthIndicator

The **GeodeContinuousQueriesHealthIndicator** provides details about registered client Continuous Queries (CQ). CQs enable client applications to receive automatic notification about events that satisfy some criteria. That criteria can be easily expressed using the predicate of an OQL query (e.g. “SELECT * FROM /Customers c WHERE c.age > 21”). Anytime data of interests is inserted or updated, and matches the criteria specified in the OQL query predicate, an event is sent to the registered client.

The following details are covered for CQs by name:

Table 11. Continuous Query(CQ) Details

Name	Description
geode.continuous-query.<name>.oql-query-string	OQL query constituting the CQ.
geode.continuous-query.<name>.closed	Indicates whether the CQ has been closed.
geode.continuous-query.<name>.closing	Indicates whether the CQ is the process of closing.
geode.continuous-query.<name>.durable	Indicates whether the CQ events will be remembered between client sessions.
geode.continuous-query.<name>.running	Indicates whether the CQ is currently running.
geode.continuous-query.<name>.stopped	Indicates whether the CQ has been stopped.

In addition, the following CQ query and statistical data is covered:

Table 12. Continuous Query(CQ), Query Details

Name	Description
geode.continuous-query.<name>.query.number-of-executions	Total number of times the query has been executed.
geode.continuous-query.<name>.query.total-execution-time	Total amount of time (ns) spent executing the query.
geode.continuous-query.<name>.statistics.number-of-deletes	

Table 13. Continuous Query(CQ), Statistic Details

Name	Description
geode.continuous-query.<name>.statistics.number-of-deletes	Number of Delete events qualified by this CQ.
geode.continuous-query.<name>.statistics.number-of-events	Total number of events qualified by this CQ.
geode.continuous-query.<name>.statistics.number-of-inserts	Number of Insert events qualified by this CQ.
geode.continuous-query.<name>.statistics.number-of-updates	Number of Update events qualified by this CQ.

In a more general sense, the Apache Geode Continuous Query system is tracked with the following, additional details on the client:

Table 14. Continuous Query(CQ), Statistic Details

Name	Description
geode.continuous-query.count	Total count of CQs.
geode.continuous-query.number-of-active	Number of currently active CQs (if available).

Name	Description
geode.continuous-query.number-of-closed	Total number of closed CQs (if available).
geode.continuous-query.number-of-created	Total number of created CQs (if available).
geode.continuous-query.number-of-stopped	Number of currently stopped CQs (if available).
geode.continuous-query.number-on-client	Number of CQs that are currently active or stopped (if available).

20.2.2. GeodePoolsHealthIndicator

The **GeodePoolsHealthIndicator** provide details about all the configured client connection **Pools**. This **HealthIndicator** primarily provides configuration meta-data for all the configured **Pools**.

The following details are covered:

Table 15. Pool Details

Name	Description
geode.pool.count	Total number of client connection Pools.
geode.pool.<name>.destroyed	Indicates whether the Pool has been destroyed.
geode.pool.<name>.free-connection-timeout	Configured amount of time to wait for a free connection from the Pool.
geode.pool.<name>.idle-timeout	The amount of time to wait before closing unused, idle connections not exceeding the configured number of minimum required connections.
geode.pool.<name>.load-conditioning-interval	Controls how frequently the Pool will check to see if a connection to a given server should be moved to a different server to improve the load balance.
geode.pool.<name>.locators	List of configured Locators.
geode.pool.<name>.max-connections	Maximum number of connections obtainable from the Pool.
geode.pool.<name>.min-connections	Minimum number of connections contained by the Pool.

Name	Description
geode.pool.<name>.multi-user-authentication	Determines whether the Pool can be used by multiple authenticated users.
geode.pool.<name>.online-locators	Returns a list of living Locators.
geode.pool.<name>.pending-event-count	Approximate number of pending subscription events maintained at server for this durable client Pool at the time it (re)connected to the server.
geode.pool.<name>.ping-interval	How often to ping the servers to verify they are still alive.
geode.pool.<name>.pr-single-hop-enabled	Whether the client will acquire a direct connection to the server containing the data of interests.
geode.pool.<name>.read-timeout	Number of milliseconds to wait for a response from a server before timing out the operation and trying another server (if any are available).
geode.pool.<name>.retry-attempts	Number of times to retry a request after timeout/exception.
geode.pool.<name>.server-group	Configures the group in which all servers this Pool connects to must belong.
geode.pool.<name>.servers	List of configured servers.
geode.pool.<name>.socket-buffer-size	Socket buffer size for each connection made in this Pool.
geode.pool.<name>.statistic-interval	How often to send client statistics to the server.
geode.pool.<name>.subscription-ack-interval	Interval in milliseconds to wait before sending acknowledgements to the cache server for events received from the server subscriptions.
geode.pool.<name>.subscription-enabled	Enabled server-to-client subscriptions.
geode.pool.<name>.subscription-message-tracking-timeout	Time-to-Live period (ms), for subscription events the client has received from the server.
geode.pool.<name>.subscription-redundancy	Redundancy level for this Pools server-to-client subscriptions, which is used to ensure clients will not miss potentially important events.

Name	Description
geode.pool.<name>.thread-local-connections	Thread local connection policy for this Pool.

20.3. Peer Cache HealthIndicators

The peer **Cache** based **HealthIndicators** provide additional details specifically for Spring Boot, peer cache member applications. These **HealthIndicators** are only available when the Spring Boot application creates a peer **Cache** instance.



The default cache instance created by Spring Boot for Apache Geode is a **ClientCache** instance.



To control what type of cache instance is created, such as a "peer", then you can explicitly declare either the **@PeerCacheApplication**, or alternatively, the **@CacheServerApplication**, annotation on your **@SpringBootApplication** annotated class.

20.3.1. GeodeCacheServersHealthIndicator

The **GeodeCacheServersHealthIndicator** provides details about the configured Apache Geode **CacheServers**. **CacheServer** instances are required to enable clients to connect to the servers in the cluster.

This **HealthIndicator** captures basic configuration meta-data and runtime behavior/characteristics of the configured **CacheServers**:

Table 16. CacheServer Details

Name	Description
geode.cache.server.count	Total number of configured CacheServer instances on this peer member.
geode.cache.server.<index>.bind-address	IP address of the NIC to which the CacheServer ServerSocket is bound (useful when the system contains multiple NICs).
geode.cache.server.<index>.hostname-for-clients	Name of the host used by clients to connect to the CacheServer (useful with DNS).
geode.cache.server.<index>.load-poll-interval	How often (ms) to query the load probe on the CacheServer.
geode.cache.server.<index>.max-connections	Maximum number of connections allowed to this CacheServer.

Name	Description
geode.cache.server.<index>.max-message-count	Maximum number of messages that can be enqueued in a client queue.
geode.cache.server.<index>.max-threads	Maximum number of Threads allowed in this CacheServer to service client requests.
geode.cache.server.<index>.max-time-between-pings	Maximum time between client pings.
geode.cache.server.<index>.message-time-to-live	Time (seconds) in which the client queue will expire.
geode.cache.server.<index>.port	Network port to which the CacheServer <code>ServerSocket</code> is bound and listening for the client connections.
geode.cache.server.<index>.running	Determines whether this CacheServer is currently running and accepting client connections.
geode.cache.server.<index>.socket-buffer-size	Configured buffer size of the Socket connection used by this CacheServer.
geode.cache.server.<index>.tcp-no-delay	Configures the TCP/IP <code>TCP_NO_DELAY</code> setting on outgoing Sockets.

In addition to the configuration settings shown above, the `CacheServer`'s `ServerLoadProbe` tracks additional details about the runtime characteristics of the `CacheServer`, as follows:

Table 17. CacheServer Metrics and Load Details

Name	Description
geode.cache.server.<index>.load.connection-load	Load on the server due to client to server connections.
geode.cache.server.<index>.load.load-per-connection	Estimate of the how much load each new connection will add to this server.
geode.cache.server.<index>.load.subscription-connection-load	Load on the server due to subscription connections.

Name	Description
geode.cache.server.<index>.load.load-per-subscription-connection	Estimate of the how much load each new subscriber will add to this server.
geode.cache.server.<index>.metrics.client-count	Number of connected clients.
geode.cache.server.<index>.metrics.max-connection-count	Maximum number of connections made to this CacheServer.
geode.cache.server.<index>.metrics.open-connection-count	Number of open connections to this CacheServer.
geode.cache.server.<index>.metrics.subscription-connection-count	Number of subscription connections to this CacheServer.

20.3.2. GeodeAsyncEventQueuesHealthIndicator

The `GeodeAsyncEventQueuesHealthIndicator` provides details about the configured `AsyncEventQueues`. AEQs can be attached to Regions to configure asynchronous, write-behind behavior.

This `HealthIndicator` captures configuration meta-data and runtime characteristics for all AEQs, as follows:

Table 18. AsyncEventQueue Details

Name	Description
geode.async-event-queue.count	Total number of configured AEQs.
geode.async-event-queue.<id>.batch-conflation-enabled	Indicates whether batch events are conflated when sent.
geode.async-event-queue.<id>.batch-size	Size of the batch that gets delivered over this AEQ.
geode.async-event-queue.<id>.batch-time-interval	Max time interval that can elapse before a batch is sent.

Name	Description
geode.async-event-queue.<id>.disk-store-name	Name of the disk store used to overflow & persist events.
geode.async-event-queue.<id>.disk-synchronous	Indicates whether disk writes are sync or async.
geode.async-event-queue.<id>.dispatcher-threads	Number of Threads used to dispatch events.
geode.async-event-queue.<id>.forward-expiration-destroy	Indicates whether expiration destroy operations are forwarded to AsyncEventListener.
geode.async-event-queue.<id>.max-queue-memory	Maximum memory used before data needs to be overflowed to disk.
geode.async-event-queue.<id>.order-policy	Order policy followed while dispatching the events to AsyncEventListeners.
geode.async-event-queue.<id>.parallel	Indicates whether this queue is parallel (higher throughput) or serial.
geode.async-event-queue.<id>.persistent	Indicates whether this queue stores events to disk.
geode.async-event-queue.<id>.primary	Indicates whether this queue is primary or secondary.
geode.async-event-queue.<id>.size	Number of entries in this queue.

20.3.3. GeodeGatewayReceiversHealthIndicator

The **GeodeGatewayReceiversHealthIndicator** provide details about the configured (WAN) **GatewayReceivers**, which are capable of receiving events from remote clusters when using Apache Geode's [multi-site, WAN topology](#).

This **HealthIndicator** captures configuration meta-data along with the running state for each **GatewayReceiver**:

Table 19. GatewayReceiver Details

Name	Description
geode.gateway-receiver.count	Total number of configured GatewayReceivers.
geode.gateway-receiver.<index>.bind-address	IP address of the NIC to which the GatewayReceiver <code>ServerSocket</code> is bound (useful when the system contains multiple NICs).
geode.gateway-receiver.<index>.end-port	End value of the port range from which the GatewayReceiver's port will be chosen.
geode.gateway-receiver.<index>.host	IP address or hostname that Locators will tell clients (i.e. GatewaySenders) that this GatewayReceiver is listening on.
geode.gateway-receiver.<index>.max-time-between-pings	Maximum amount of time between client pings.
geode.gateway-receiver.<index>.port	Port on which this GatewayReceiver listens for clients (i.e. GatewaySenders).
geode.gateway-receiver.<index>.running	Indicates whether this GatewayReceiver is running and accepting client connections (from GatewaySenders).
geode.gateway-receiver.<index>.socket-buffer-size	Configured buffer size for the Socket connections used by this GatewayReceiver.
geode.gateway-receiver.<index>.start-port	Start value of the port range from which the GatewayReceiver's port will be chosen.

20.3.4. GeodeGatewaySendersHealthIndicator

The `GeodeGatewaySendersHealthIndicator` provides details about the configured `GatewaySenders`. `GatewaySenders` are attached to Regions in order to send Region events to remote clusters in Apache Geode's [multi-site, WAN topology](#).

This `HealthIndicator` captures essential configuration meta-data and runtime characteristics for each `GatewaySender`:

Table 20. GatewaySender Details

Name	Description
geode.gateway-sender.count	Total number of configured GatewaySenders.

Name	Description
geode.gateway-sender.<id>.alert-threshold	Alert threshold (ms) for entries in this GatewaySender's queue.
geode.gateway-sender.<id>.batch-conflation-enabled	Indicates whether batch events are conflated when sent.
geode.gateway-sender.<id>.batch-size	Size of the batches sent.
geode.gateway-sender.<id>.batch-time-interval	Max time interval that can elapse before a batch is sent.
geode.gateway-sender.<id>.disk-store-name	Name of the DiskStore used to overflow and persist queue events.
geode.gateway-sender.<id>.disk-synchronous	Indicates whether disk writes are sync or async.
geode.gateway-sender.<id>.dispatcher-threads	Number of Threads used to dispatch events.
geode.gateway-sender.<id>.max-queue-memory	Maximum amount of memory (MB) usable for this GatewaySender's queue.
geode.gateway-sender.<id>.max-parallelism-for-replicated-region	
geode.gateway-sender.<id>.order-policy	Order policy followed while dispatching the events to GatewayReceivers.
geode.gateway-sender.<id>.parallel	Indicates whether this GatewaySender is parallel (higher throughput) or serial.
geode.gateway-sender.<id>.paused	Indicates whether this GatewaySender is paused.
geode.gateway-sender.<id>.persistent	Indicates whether this GatewaySender persists queue events to disk.

Name	Description
geode.gateway-sender.<id>.remote-distributed-system-id	Identifier for the remote distributed system.
geode.gateway-sender.<id>.running	Indicates whether this GatewaySender is currently running.
geode.gateway-sender.<id>.socket-buffer-size	Configured buffer size for the Socket connections between this GatewaySender and its receiving GatewayReceiver.
geode.gateway-sender.<id>.socket-read-timeout	Amount of time (ms) that a Socket read between this sending GatewaySender and its receiving GatewayReceiver will block.

Chapter 21. Spring Session

This chapter covers auto-configuration of Spring Session using Apache Geode to manage (HTTP) Session state in a reliable (consistent), highly-available (replicated) and clustered manner.

[Spring Session](#) provides an API and several implementations for managing a user's session information. It has the ability to replace the `javax.servlet.http.HttpSession` in an application container neutral way along with providing Session IDs in HTTP headers to work with RESTful APIs.

Furthermore, Spring Session provides the ability to keep the `HttpSession` alive even when working with WebSockets and reactive Spring WebFlux WebSessions.

A full discussion of Spring Session is beyond the scope of this document, and the reader is encouraged to learn more by reading the [docs](#) and reviewing the [samples](#).

Of course, Spring Boot for Apache Geode provides auto-configuration support to configure Apache Geode as the user's session information management provider and store when [Spring Session for Apache Geode](#) is on your Spring Boot application's classpath.



You can learn more about Spring Session for Apache Geode in the [docs](#).



Refer to the corresponding Sample [Guide](#) and [Code](#) to see Spring Session for Apache Geode in action!

21.1. Configuration

There is nothing special that you need to do in order to use Apache Geode as a Spring Session provider, managing the (HTTP) Session state of your Spring Boot application.

Simply include the appropriate Spring Session dependency on your Spring Boot application's classpath, for example:

Maven dependency declaration

```
<dependency>
  <groupId>org.springframework.session</groupId>
  <artifactId>spring-session-data-geode</artifactId>
  <version>2.5.0-RC1</version>
</dependency>
```

Alternatively, you may declare the provided `spring-geode-starter-session` dependency in your Spring Boot application Maven POM or Gradle build file:

Maven dependency declaration

```
<dependency>
  <groupId>org.springframework.geode</groupId>
  <artifactId>spring-geode-starter-session</artifactId>
  <version>1.5.0-RC1</version>
</dependency>
```

After declaring the required Spring Session dependency, then begin your Spring Boot application as you normally would:

Spring Boot Application

```
@SpringBootApplication
public class MySpringBootApplication {

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

    // ...
}
```

That is it!

Of course, you are free to create application-specific, Spring Web MVC **Controllers** to interact with the **HttpSession** as needed by your application:

Application Controller using HttpSession

```
@Controller
class MyApplicationController {

    @GetRequest("...")
    public String processGet(HttpSession session) {
        // interact with HttpSession
    }
}
```

The **HttpSession** is replaced by a Spring managed **Session** that will be stored in Apache Geode.

21.2. Custom Configuration

By default, Spring Boot for Apache Geode (SBDG) applies reasonable and sensible defaults when configuring Apache Geode as the provider in Spring Session.

So, for instance, by default, SBDG set the session expiration timeout to 30 minutes. It also uses a **ClientRegionShortcut.PROXY** as the client Region data management policy for the Apache Geode Region managing the (HTTP) Session state when the Spring Boot application is using a **ClientCache**,

which it does by [default](#).

However, what if the defaults are not sufficient for your application requirements?

21.2.1. Custom Configuration using Properties

Spring Session for Apache Geode publishes [well-known configuration properties](#) for each of the various Spring Session configuration options when using Apache Geode as the (HTTP) Session state management provider.

You may specify any of these properties in a Spring Boot `application.properties` file to adjust Spring Session's configuration when using Apache Geode.

In addition to the properties provided in and by Spring Session for Apache Geode, Spring Boot for Apache Geode also recognizes and respects the `spring.session.timeout` property as well as the `server.servlet.session.timeout` property as discussed [here](#).



`spring.session.data.gemfire.session.expiration.max-inactive-interval-seconds` takes precedence over `spring.session.timeout`, which takes precedence over `server.servlet.session.timeout`, when any combination of these properties have been simultaneously configured in the Spring `Environment` of your application.

21.2.2. Custom Configuration using a Configurer

Spring Session for Apache Geode also provides the `SpringSessionGemFireConfigurer` callback interface, which can be declared in your Spring `ApplicationContext` to programmatically control the configuration of Spring Session when using Apache Geode.

The `SpringSessionGemFireConfigurer`, when declared in the Spring `ApplicationContext`, takes precedence over any of the Spring Session (for Apache Geode) configuration properties, and will effectively override them when both are present.

More information on using the `SpringSessionGemFireConfigurer` can be found in the [docs](#).

21.3. Disabling Session State Caching

There may be cases where you do not want your Spring Boot application to manage (HTTP) Session state using Apache Geode. In certain cases, you may be using another Spring Session provider, such as Redis, to cache and manage your Spring Boot application's (HTTP) Session state, while, even in other cases, you do not want to use Spring Session to manage your (HTTP) Session state at all. Rather, you prefer to use your Web Server's (e.g. Tomcat) `HttpSession` state management.

Either way, you can specifically call out your Spring Session provider using the `spring.session.store-type` property in `application.properties`, as follows:

```
#application.properties  
  
spring.session.store-type=redis  
...
```

If you prefer not to use Spring Session to manage your Spring Boot application's (HTTP) Session state at all, then do the following:

Use Web Server Session State Management

```
#application.properties  
  
spring.session.store-type=none  
...
```

Again, see Spring Boot [docs](#) for more details.



It is possible to include multiple providers on the classpath of your Spring Boot application. For instance, you might be using Redis to cache your application's (HTTP) Session state while using Apache Geode as your application's persistent store (*System of Record*).



Spring Boot does not properly recognize `spring.session.store-type=[gemfire|geode]` even though Spring Boot for Apache Geode is setup to handle either of these property values (i.e. either “gemfire” or “geode”).

21.4. Using Spring Session with Pivotal Cloud Cache (PCC)

Whether you are using Spring Session in a Spring Boot `ClientCache` application connecting to an externally managed cluster of Apache Geode servers, or connecting to a cluster of servers in a Pivotal Cloud Cache service instance managed by a VMware Tanzu Application Service (TAS) environment, the setup is the same.

Spring Session for Apache Geode expects there to exist a cache Region in the cluster that will store and manage (HTTP) Session state when your Spring Boot application is a `ClientCache` application in a client/server topology.

By default, the cache Region used to store and manage (HTTP) Session state is called “*ClusteredSpringSessions*”.

You can set the name of the cache Region used to store and manage (HTTP) Session state either by explicitly declaring the `@EnableGemFireHttpSession` annotation on your main `@SpringBootApplication` class, like so:

Using `@EnableGemfireHttpSession`

```
@SpringBootApplication
@EnableGemFireHttpSession(regionName = "MySessions")
class MySpringBootSpringSessionApplication {
    // ...
}
```

Or alternatively, we recommend users to configure the cache Region name using the well-known and documented property in Spring Boot `application.properties`:

Using properties

```
spring.session.data.gemfire.session.region.name=MySessions
```

Once you decide on the cache Region name used to store and manage (HTTP) Sessions, you must create the Region in the cluster somehow.

On the client, this is simple since SBDG's auto-configuration will automatically create the client `PROXY` Region used to send/receive (HTTP) Session state between the client and server for you, when either Spring Session is on the application classpath (e.g. `spring-geode-starter-session`), or you explicitly declare the `@EnableGemFireHttpSession` annotation on your main `@SpringBootApplication` class.

However, on the server-side, you currently have a couple of options.

First, you can create the cache Region manually using `Gfsh`, like so:

Create the Sessions Region using `Gfsh`

```
gfsh> create region --name=MySessions --type=PARTITION --entry-idle-time
-expiration=1800
      --entry-idle-time-expiration-action=INVALIDATE
```

You must create the cache Region with the appropriate name and an expiration policy.

In this case, we created an Idle Expiration Policy with a timeout of `1800 seconds (30 minutes)`, after which, the entry (i.e. Session object) will be `"invalidated"`.



Session expiration is managed by the Expiration Policy set on the cache Region used to store Session state. The Servlet Container's (HTTP) Session expiration configuration is not used since Spring Session is replacing the Servlet Container's Session management capabilities with its own and Spring Session delegates this behavior to the individual providers, like Apache Geode.

Alternatively, you could send the definition for the cache Region from your Spring Boot `ClientCache` application to the cluster using the SBDG `@EnableClusterAware` annotation, which is meta-annotated with SDG's `@EnableClusterConfiguration` annotation.



See the [Javadoc](#) on the `@EnableClusterConfiguration` annotation as well as the [documentation](#) for more details.

Using `@EnableClusterAware`

```
@SpringBootApplication
@EnableClusterAware
class MySpringBootSpringSessionApplication {
    // ...
}
```

However, it is not currently possible to send Expiration Policy configuration metadata to the cluster yet. Therefore, you must manually alter the cache Region to set the Expiration Policy, like so:

Using *Gfsh* to Alter Region

```
gfsh> alter region --name=MySessions --entry-idle-time-expiration=1800
      --entry-idle-time-expiration-action=INVALIDATE
```

That is it!

Now your Spring Boot `ClientCache` application using Spring Session in a client/server topology is configured to store and manage user (HTTP) Session state in the cluster. This works for either standalone, externally managed Apache Geode clusters, or when using PCC running in a VMware Tanzu Application Service environment.

Chapter 22. Pivotal CloudFoundry



As of the VMware, Inc acquisition of Pivotal Software, Inc, Pivotal CloudFoundry (PCF) is now known as VMware Tanzu Application Service (TAS) for VMs. Also, Pivotal Cloud Cache (PCC) has been rebranded as VMware Tanzu GemFire for VMS. This documentation will eventually be updated to reflect the rebranding.

In most cases, when you deploy (i.e. "*push*") your Spring Boot applications to Pivotal CloudFoundry (PCF) you will bind your app to 1 or more instances of the Pivotal Cloud Cache (PCC) service.

In a nutshell, [Pivotal Cloud Cache](#) (PCC) is a managed version of [Pivotal GemFire](#) running in [Pivotal CloudFoundry](#) (PCF). When running in or across cloud environments (e.g. AWS, Azure, GCP or PWS), PCC with PCF offers several advantages over trying to run and manage your own standalone Apache Geode clusters. It handles many of the infrastructure-related, operational concerns so you do not have to.

22.1. Running Spring Boot applications as a specific user

By default, Spring Boot applications run as a "*cluster_operator*" Role-based user in Pivotal CloudFoundry when the app is bound to a Pivotal Cloud Cache service instance.

A "*cluster_operator*" has full system privileges (i.e. Authorization) to do whatever that user wishes to involving the PCC service instance. A "*cluster_operator*" has read/write access to all the data, can modify the schema (e.g. create/destroy Regions, add/remove Indexes, change eviction or expiration policies, etc), start and stop servers in the PCC cluster, or even modify permissions.

About *cluster-operator* as the default user

1 of the reasons why Spring Boot apps default to running as a "*cluster_operator*" is to allow configuration metadata to be sent from the client to the server. Enabling configuration metadata to be sent from the client to the server is a useful development-time feature and is as simple as annotating your main `@SpringBootApplication` class with the `@EnableClusterConfiguration` annotation:

Using `@EnableClusterConfiguration`

```
@SpringBootApplication
@EnableClusterConfiguration(useHttp = true)
class SpringBootApacheGeodeClientCacheApplication { }
```

With `@EnableClusterConfiguration`, Region and OQL Index configuration metadata defined on the client can be sent to servers in the PCC cluster. Apache Geode requires matching Regions by name on both the client and servers in order for clients to send and receive data to and from the cluster.

For example, when you declare the Region where an application entity will be persisted using the `@Region` mapping annotation and additionally declare the `@EnableEntityDefinedRegions` annotation on the main `@SpringBootApplication` class in conjunction with the `@EnableClusterConfiguration` annotation, then not only will SBDG create the required client Region, but it will also send the configuration metadata for this Region to the servers in the cluster to create the matching, required server Region, where the data for your application entity will be managed.

However...

With great power comes great responsibility. - Uncle Ben

Not all Spring Boot applications using PCC will need to change the schema, or even modify data. Rather, certain apps may only need read access. Therefore, it is ideal to be able to configure your Spring Boot applications to run with a different user at runtime other than the auto-configured "*cluster_operator*", by default.

A prerequisite for running a Spring Boot application using PCC with a specific user is to create a user with restricted permissions using Pivotal CloudFoundry *AppsManager* while provisioning the PCC service instance to which the Spring Boot app will be bound.

Configuration metadata for the PCC service instance might appear as follows:


```
{
  "p-cloudcache": [{
    "credentials": {
      "distributed_system_id": "0",
      "locators": [ "localhost[55221]" ],
      "urls": {
        "gfsh": "https://cloudcache-12345.services.cf.pws.com/gemfire/v1",
        "pulse": "https://cloudcache-12345.services.cf.pws.com/pulse"
      },
      "users": [{
        "password": "*****",
        "roles": [ "cluster_operator" ],
        "username": "cluster_operator_user"
      }, {
        "password": "*****",
        "roles": [ "developer" ],
        "username": "developer_user"
      }, {
        "password": "*****",
        "roles": [ "read-only-user" ],
        "username": "guest"
      }],
      "wan": {
        "sender_credentials": {
          "active": {
            "password": "*****",
            "username": "gateway-sender-user"
          }
        }
      }
    },
    "name": "jblum-pcc",
    "plan": "small",
    "tags": [ "gemfire", "cloudcache", "database", "pivotal" ]
  }]
}
```

In the PCC service instance configuration metadata above, we see a *guest* user with the *read-only-user* Role. If the *read-only-user* Role is properly configured with *read-only* permissions as the name implies, then we could configure our Spring Boot application to run as *guest* with read-only access using:

Configuring a Spring Boot app to run as a specific user

```
# Spring Boot application.properties for PCF when using PCC

spring.data.gemfire.security.username=guest
```



The `spring.data.gemfire.security.username` property corresponds directly to the SDG `@EnableSecurity` annotation, `securityUsername` attribute. See the [Javadoc](#) for more details.

The `spring.data.gemfire.security.username` property is the same property used by Spring Data for Apache Geode (SDG) to configure the runtime user of your Spring Data application when connecting to an externally managed Apache Geode cluster.

In this case, SBDG simply uses the configured username to lookup the authentication credentials of the user to set the username and password used by the Spring Boot, `ClientCache` app when connecting to PCC while running in PCF.

If the username is not valid, then an `IllegalStateException` is thrown.

By using [Spring Profiles](#), it would be a simple matter to configure the Spring Boot application to run with a different user depending on environment.

See the Pivotal Cloud Cache documentation on [Security](#) for configuring users with assigned roles & permissions.

22.1.1. Overriding Authentication Auto-configuration

It should be generally understood that *auto-configuration* for client authentication is only available for managed environments, like Pivotal CloudFoundry. When running in externally managed environments, you must explicitly set a username and password to authenticate, as described [here](#).

To completely override the *auto-configuration* of client authentication, simply set both a username and password:

Overriding Security Authentication Auto-configuration with explicit username and password

```
# Spring Boot application.properties

spring.data.gemfire.security.username=MyUser
spring.data.gemfire.security.password=MyPassword
```

In this case, SBDG's *auto-configuration* for authentication is effectively disabled and security credentials will not be extracted from the environment.

22.2. Targeting Specific Pivotal Cloud Cache Service Instances

It is possible to provision multiple instances of the Pivotal Cloud Cache service in your Pivotal CloudFoundry environment. You can then bind multiple PCC service instances to your Spring Boot app.

However, Spring Boot for Apache Geode (SBDG) will only auto-configure 1 PCC service instance for your Spring Boot application. This does not mean it is not possible to use multiple PCC service

instances with your Spring Boot app, just that SBDG only "auto-configures" 1 service instance for you.

You must select which PCC service instance your Spring Boot app will auto-configure for you automatically when you have multiple instances and want to target a specific PCC service instance to use.

To do so, declare the following SBDG property in Spring Boot `application.properties`:

Spring Boot application.properties targeting a specific PCC service instance by name

```
# Spring Boot application.properties

spring.boot.data.gemfire.cloud.cloudfoundry.service.cloudcache.name=pccServiceInstanceTwo
```

The `spring.boot.data.gemfire.cloud.cloudfoundry.service.cloudcache.name` property tells SBDG which PCC service instance to auto-configure.

If the named PCC service instance identified by the property does not exist, then SBDG will throw an `IllegalStateException` stating the PCC service instance by name could not be found.

If you did not set the property and your Spring Boot app is bound to multiple PCC service instances, then SBDG will auto-configure the first PCC service instance it finds by name, alphabetically.

If you did not set the property and no PCC service instance is found, then SBDG will log a warning.

22.3. Using Multiple Pivotal Cloud Cache Service Instances

If you want to use multiple PCC service instances with your Spring Boot application, then you need to configure multiple connection `Pools` connected to each PCC service instance used by your Spring Boot application.

The configuration would be similar to the following:

Multiple Pivotal Cloud Cache Service Instance Configuration

```
@Configuration
@EnablePools(pools = {
    @EnablePool(name = "PccOne"),
    @EnablePool(name = "PccTwo"),
    ...,
    @EnablePool(name = "PccN")
})
class PccConfiguration {
    // ...
}
```

You would then externalize the configuration for the individually declared **Pools** in Spring Boot `application.properties`:

Configuring Pool Locator connection endpoints

```
# Spring Boot `application.properties`

spring.data.gemfire.pool.pccone.locators=pccOneHost1[port1], pccOneHost2[port2], ...,
pccOneHostN[portN]

spring.data.gemfire.pool.pcctwo.locators=pccTwoHost1[port1], pccTwoHost2[port2], ...,
pccTwoHostN[portN]
```



Though less common, you can also configure the **Pool** of connections to target specific servers in the cluster using the `spring.data.gemfire.pool.<named-pool>.severs` property.



Keep in mind that properties in Spring Boot `application.properties` can refer to other properties like so: `property=${otherProperty}`. This allows you to further externalize properties using Java System properties or Environment Variables.

Of course, a client Region is then assigned the Pool of connections that are used to send data to/from the specific PCC service instance (cluster):

Assigning a Pool to a client Region

```
@Configuration
class GeodeConfiguration {

    @Bean("Example")
    ClientRegionFactoryBean exampleRegion(GemFireCache gemfireCache,
        @Qualifier("PccTwo") Pool poolForPccTwo) {

        ClientRegionFactoryBean exampleRegion = new ClientRegionFactoryBean();

        exampleRegion.setCache(gemfireCache);
        exampleRegion.setPool(poolForPccTwo);
        exampleRegion.setShortcut(ClientRegionShortcut.PROXY);

        return exampleRegion;
    }
}
```

You can configure as many Pools and client Regions as needed by your application. Again, the **Pool** determines which Pivotal Cloud Cache service instance and cluster the data for the client Region will reside.



By default, SBDG configures all **Pools** declared in a Spring Boot, **ClientCache** application to connect to and use a single PCC service instance. This may be a targeted PCC service instance when using the `spring.boot.data.gemfire.cloud.cloudfoundry.service.cloudcache.name` property as discussed [above](#).

22.4. Hybrid Pivotal CloudFoundry & Apache Geode Spring Boot Applications

Sometimes, it is desirable to deploy (i.e. "push") and run your Spring Boot applications in Pivotal CloudFoundry, but still connect your Spring Boot applications to an externally managed, standalone Apache Geode cluster.

Spring Boot for Apache Geode (SBDG) makes this a non-event and honors its *"little to no code or configuration changes necessary"* goal, regardless of your runtime choice, *"it should just work!"*

To help guide you through this process, we will cover the following topics:

1. Install and Run PCFDev.
2. Start an Apache Geode cluster.
3. Create a User-Provided Service (CUPS).
4. Push and Bind a Spring Boot application.
5. Run the Spring Boot application.

22.4.1. Running PCFDev

For this exercise, we will be using [PCF Dev](#).

PCF Dev, much like PCF, is an elastic application runtime for deploying, running and managing your Spring Boot applications. However, it does so in the confines of your local development environment, i.e. your workstation.

Additionally, PCF Dev provides several services out-of-the-box, such as MySQL, Redis and RabbitMQ. These services can be bound and used by your Spring Boot application to accomplish its tasks.

However, PCF Dev lacks the Pivotal Cloud Cache service that is available in PCF. This is actually ideal for this little exercise since we are trying to build and run Spring Boot applications in a PCF environment but connect to an externally managed, standalone Apache Geode cluster.

As a prerequisite, you will need to follow the steps outlined in the [tutorial](#) to get PCF Dev setup and running on your workstation.

To run PCF Dev, you will execute the following **cf** CLI command, replacing the path to the TGZ file with the file you acquired from the [download](#):

Start PCF Dev

```
$ cf dev start -f ~/Downloads/Pivotal/CloudFoundry/Dev/pcfdev-v1.2.0-darwin.tgz
```

You should see output similar to:

Running PCF Dev

[illegible]

To use the **cf** CLI tool, you must login to the PCF Dev environment:

Login to PCF Dev using **cf** CLI

```
$ cf login -a https://api.dev.cfdev.sh --skip-ssl-validation
```

You can also access the [PCF Dev Apps Manager](https://apps.dev.cfdev.sh/) tool from your Web browser at the following URL:

apps.dev.cfdev.sh/

Apps Manager provides a nice UI to manage your org, space, services and apps. It lets you push and update apps, create services, bind apps to the services and start and stop your deployed applications, among many other things.

22.4.2. Running an Apache Geode Cluster

Now that PCF Dev is setup and running, we need to start an external, standalone Apache Geode cluster that our Spring Boot application will connect to and use to manage its data.

You will need to install a [distribution](#) of Apache Geode on your workstation. Then you must set the **\$GEODE** environment variable. It is also convenient to add **\$GEODE/bin** to your system **\$PATH**.

Afterward, you can launch the Geode Shell (*Gfsh*) tool:

Running Gfsh

```
$ echo $GEODE
/Users/jblum/pivdev/apache-geode-1.6.0

$ gfsh

-----
 /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /  _ _ _ _ /
/ /  _ _ /  _ _ /  _ _ /  _ _ _ _ /
/ /  _ _ /  _ _ /  _ _ /  _ _ /  _ _ /
/ _ _ _ _ /  _ _ _ _ /  _ _ _ _ /  _ _ 1.6.0

Monitor and Manage Apache Geode
gfsh>
```

We have conveniently provided the *Gfsh* shell script used to start the Apache Geode cluster:

Gfsh shell script to start the Apache Geode cluster

```
#!/bin/gfsh
# Gfsh shell script to configure and bootstrap an Apache Geode cluster.

start locator --name=LocatorOne --log-level=config --classpath=@project-dir@/apache
-geode-extensions/build/libs/apache-geode-extensions-@project-version@.jar --J=
-Dgemfire.security-manager=org.springframework.geode.security.TestSecurityManager --J=
-Dgemfire.http-service-port=8080

start server --name=ServerOne --log-level=config --user=admin --password=admin
--classpath=@project-dir@/apache-geode-extensions/build/libs/apache-geode-extensions
-@project-version@.jar
```

The `start-cluster.gfsh` shell script starts one Geode Locator and one Geode Server.

A Locator is used by clients to discover and connect to servers in the cluster to manage its data. A Locator is also used by new servers joining a cluster as a peer member, which allows the cluster to be elastically scaled-out (or scaled-down, as needed). A Geode Server stores the data for the application.

You can start as many Locators or Servers as necessary to meet the availability and load demands of your application. Obviously, the more Locators and Servers your cluster has, the more resilient it is to failure. However, you should size your cluster accordingly, based on your application's needs since there is overhead relative to the cluster size.

You will see output similar to the following when starting the Locator and Server:

Starting the Apache Geode cluster

```
gfsh>start locator --name=LocatorOne --log-level=config
--classpath=/Users/jblum/pivdev/spring-boot-data-geode/apache-geode
-extensions/build/libs/apache-geode-extensions-1.1.0.BUILD-SNAPSHOT.jar --J=
-Dgemfire.security-manager=org.springframework.geode.security.TestSecurityManager --J=
-Dgemfire.http-service-port=8080
Starting a Geode Locator in /Users/jblum/pivdev/lab/LocatorOne...
..
Locator in /Users/jblum/pivdev/lab/LocatorOne on 10.99.199.24[10334] as LocatorOne is
currently online.
Process ID: 14358
Uptime: 1 minute 1 second
Geode Version: 1.6.0
Java Version: 1.8.0_192
Log File: /Users/jblum/pivdev/lab/LocatorOne/LocatorOne.log
JVM Arguments: -Dgemfire.enable-cluster-configuration=true -Dgemfire.load-cluster
-configuration-from-dir=false -Dgemfire.log-level=config -Dgemfire.security
-manager=org.springframework.geode.security.TestSecurityManager -Dgemfire.http-service
-port=8080 -Dgemfire.launcher.registerSignalHandlers=true -Djava.awt.headless=true
-Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /Users/jblum/pivdev/apache-geode-1.6.0/lib/geode-core-
```



```
1.6.0.jar:/Users/jblum/pivdev/spring-boot-data-geode/apache-geode-extensions/build/libs/apache-geode-extensions-1.1.0.BUILD-SNAPSHOT.jar:/Users/jblum/pivdev/apache-geode-1.6.0/lib/geode-dependencies.jar
```

Security Manager is enabled - unable to auto-connect. Please use "connect --locator=10.99.199.24[10334] --user --password" to connect Gfsh to the locator.

Authentication required to connect to the Manager.

```
gfsh>connect
```

```
Connecting to Locator at [host=localhost, port=10334] ..
```

```
Connecting to Manager at [host=10.99.199.24, port=1099] ..
```

```
user: admin
```

```
password: *****
```

```
Successfully connected to: [host=10.99.199.24, port=1099]
```

```
gfsh>start server --name=ServerOne --log-level=config --user=admin --password=admin
```

```
--classpath=/Users/jblum/pivdev/spring-boot-data-geode/apache-geode
```

```
-extensions/build/libs/apache-geode-extensions-1.1.0.BUILD-SNAPSHOT.jar
```

```
Starting a Geode Server in /Users/jblum/pivdev/lab/ServerOne...
```

```
....
```

```
Server in /Users/jblum/pivdev/lab/ServerOne on 10.99.199.24[40404] as ServerOne is currently online.
```

```
Process ID: 14401
```

```
Uptime: 3 seconds
```

```
Geode Version: 1.6.0
```

```
Java Version: 1.8.0_192
```

```
Log File: /Users/jblum/pivdev/lab/ServerOne/ServerOne.log
```

```
JVM Arguments: -Dgemfire.default.locators=10.99.199.24[10334] -Dgemfire.security
```

```
-username=admin -Dgemfire.start-dev-rest-api=false -Dgemfire.security
```

```
-password=***** -Dgemfire.use-cluster-configuration=true -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/jblum/pivdev/apache-geode-1.6.0/lib/geode-core-
```

```
1.6.0.jar:/Users/jblum/pivdev/spring-boot-data-geode/apache-geode-
```

```
extensions/build/libs/apache-geode-extensions-1.1.0.BUILD-
```

```
SNAPSHOT.jar:/Users/jblum/pivdev/apache-geode-1.6.0/lib/geode-dependencies.jar
```

Once the cluster has been started successfully, you can list the members:

List members of the cluster

```
gfsh>list members
```

Name	Id
LocatorOne	10.99.199.24(LocatorOne:14358:locator)<ec><v0>:1024 [Coordinator]
ServerOne	10.99.199.24(ServerOne:14401)<v1>:1025

Currently, we have not defined any Regions in which to store our application's data:

```
gfsh>list regions
No Regions Found
```

This is deliberate since we are going to let the application drive its schema structure, both on the client (app) as well as on the server-side (cluster). More on this below.

22.4.3. Creating a User-Provided Service

Now that we have PCF Dev and a small Apache Geode cluster up and running, it is time to create a User-Provided Service to the external, standalone Apache Geode cluster that we started in [step 2](#).

As mentioned, PCF Dev offers the MySQL, Redis and RabbitMQ services out-of-the-box. However, to use Apache Geode in the same capacity as you would Pivotal Cloud Cache when running in a production-grade, PCF environment, you need to create a User-Provided Service for the standalone Apache Geode cluster.

To do so, execute the following **cf** CLI command:

cf cups command

```
$ cf cups <service-name> -t "gemfire, cloudcache, database, pivotal" -p '<service-credentials-in-json>'
```



It is important that you specify the tags ("gemfire, cloudcache, database, pivotal") exactly as shown in the **cf** CLI command above.

The argument passed to the **-p** command-line option is a JSON document (object) containing the "credentials" for our User-Provided Service.

The JSON object is as follows:

User-Provided Service Credentials JSON

```
{
  "locators": [ "<hostname>[<port>]" ],
  "urls": { "gfsh": "https://<hostname>/gemfire/v1" },
  "users": [{ "password": "<password>", "roles": [ "cluster_operator" ], "username":
"<username>" }]
}
```

The complete **cf** CLI command would be similar to the following:

Example `cf cups` command

```
cf cups apacheGeodeService -t "gemfire, cloudcache, database, pivotal" \
  -p '{ "locators": [ "10.99.199.24[10334]" ], "urls": { "gfsh":
  "https://10.99.199.24/gemfire/v1" }, "users": [{ "password": "admin", "roles": [
  "cluster_operator" ], "username": "admin" }] }'
```

We replaced the `<hostname>` placeholder tag with the IP address of our external Apache Geode Locator. The IP address can be found in the *Gfsh start locator* output above.

Additionally, the `<port>` placeholder tag has been replaced with the default Locator port, `10334`,

Finally, we set the `username` and `password` accordingly.



Spring Boot for Apache Geode (SBDG) provides template files in the `/opt/jenkins/data/workspace/spring-boot-data-geode_master/spring-geode-docs/src/main/resources` directory.

Once the service has been created, you can query the details from the `cf` CLI:

```
$ cf services
Getting services in org cfdev-org / space cfdev-space as admin...

name                service          plan    bound apps    last operation    broker
apacheGeodeService  user-provided          boot-pcc-demo

$ cf service apacheGeodeService
Showing info of service apacheGeodeService in org cfdev-org / space cfdev-space as
admin...

name:      apacheGeodeService
service:   user-provided
tags:      gemfire, cloudcache, database, pivotal

bound apps:
name        binding name    status      message
boot-pcc-demo                create succeeded
```

You can also view the "apacheGeodeService" from Apps Manager, starting from the `Service` tab in your org and space:

Pivotal Apps Manager

Search apps, services, spaces, & orgs

press **J** admin

Home

Marketplace

Accounting Report

Home / cfdev-org / cfdev-space

SPACE


cfdev-space

RUNNING 0 STOPPED 1 CRASHED 0

App (1) Service (1) Route (1) Members (2) Settings

Services

ADD A SERVICE

Service	Name	Bound Apps	Plan	Last Operation
 User Provided	apacheGeodeService	1	User Provided	

By clicking on the "apacheGeodeService" service entry in the table you can get all the service details, such the bound apps:

Pivotal Apps Manager

Search apps, services, spaces, & orgs


press **J** admin

Home

Marketplace

Accounting Report

Home / cfdev-org / cfdev-space / apacheGeodeService

 **apacheGeodeService**
SERVICE: User Provided

Overview Configuration Settings

Bound Apps

BIND APP

boot-pcc-demo

Bound Routes

This service does not support route binding.

Configuration:

Pivotal Apps Manager

Search apps, services, spaces, & orgs


press **J** admin

Home

Marketplace

Accounting Report

Home / cfdev-org / cfdev-space / apacheGeodeService

 **apacheGeodeService**
SERVICE: User Provided

Overview Configuration Settings

Configuration

Credential Parameters (Optional) Enter JSON ☒

JSON

```
1 [{"locators":["10.99.199.24[10334]"],"urls":{"gfsh":"https://10.99.199.24:8080/gemfire/v1"},"users":[{"password":"admin","roles":["cluster_operator"],"username":"
```

Syslog Drain Url (Optional)

Route Service Url (Optional)

Docs

Tools

<https://apps.dev.cfdev.sh>

CANCEL UPDATE SERVICE

And so on.



You can learn more about CUPS in the PCF documentation, [here](#).

22.4.4. Push & Bind a Spring Boot application

Now it is time to push a Spring Boot application to PCF Dev and bind the app to the "apacheGeodeService".

Any Spring Boot **ClientCache** application using SBDG will do. For this example, we will use the **PCCDemo** application, available in *GitHub*.

After cloning the project to your workstation, you must perform a build to produce the artifact to push to PCF Dev:

Build the PCCDemo app

```
$ mvn clean package
```

Then, you can push the app to PCF Dev with the following **cf** CLI command:

Push app to PCF Dev

```
$ cf push boot-pcc-demo -u none --no-start -p target/client-0.0.1-SNAPSHOT.jar
```

Once the app has been successfully deployed to PCF Dev, you can get app details:

Details for deployed app

```
$ cf apps
Getting apps in org cfdev-org / space cfdev-space as admin...
OK

name            requested state  instances  memory  disk  urls
boot-pcc-demo   stopped         0/1        768M    1G    boot-pcc-
demo.dev.cfdev.sh

$ cf app boot-pcc-demo
Showing health and status for app boot-pcc-demo in org cfdev-org / space cfdev-space
as admin...

name:            boot-pcc-demo
requested state:  stopped
routes:          boot-pcc-demo.dev.cfdev.sh
last uploaded:   Tue 02 Jul 00:34:09 PDT 2019
stack:           cflinuxfs3
buildpacks:      https://github.com/cloudfoundry/java-buildpack.git

type:            web
instances:       0/1
memory usage:    768M

  state  since                cpu  memory  disk  details
#0  down   2019-07-02T21:48:25Z  0.0%  0 of 0  0 of 0

type:            task
instances:       0/0
memory usage:    256M

There are no running instances of this process.
```

You can either bind the PPCDemo app to the "apacheGeodeService" using the **cf** CLI command:

Bind app to apacheGeodeService using CLI

```
cf bind-service boot-pcc-demo apacheGeodeService
```

Or, alternatively, you can create a YAML file (**manifest.yml** in **src/main/resources**) containing the deployment descriptor:

Example YAML deployment descriptor file

```
\---
applications:
  - name: boot-pcc-demo
    memory: 768M
    instances: 1
    path: ./target/client-0.0.1-SNAPSHOT.jar
    services:
      - apacheGeodeService
  buildpacks:
    - https://github.com/cloudfoundry/java-buildpack.git
```

You can also use Apps Manager to view app details and un/bind additional services. Start by navigating to the **App** tab under your org and space:

The screenshot shows the Pivotal Apps Manager interface. The left sidebar contains links for Home, Marketplace, and Accounting Report. The top navigation bar includes a search bar and a user profile. The main content area shows the 'App (1)' tab for the 'boot-pcc-demo' app. A table lists the app's status as 'Stopped' with 1 instance, 768 MB memory, and a route link.

Status	Name	Instances	Memory	Last Push	Route
Stopped	boot-pcc-demo	1	768 MB	14 hours ago	https://boot-pcc-demo.dev.cfdev.sh

From there, you can click on the desired app and navigate to the **Overview**:

The screenshot shows the 'Overview' tab for the 'boot-pcc-demo' app. The left sidebar contains links for Home, Marketplace, and Accounting Report. The top navigation bar includes a search bar and a user profile. The main content area shows the 'Overview' tab with a list of events and a summary of the app's configuration.

Events

- Stopped app
admin 07/02/2019 at 11:49:19 AM
- Started app
admin 07/02/2019 at 12:34:17 AM
- Mapped route to app
admin 07/02/2019 at 12:32:15 AM
- Created app
admin 07/02/2019 at 12:32:15 AM

App Summary

Instances / Allocated	Memory / Allocated	Disk / Allocated
0 / 0	0.00 / 0.75 GB	0.00 / 1.00 GB

Processes and Instances

Process	Instances	Memory Allocated	Disk Allocated	Scale
web	0	768 MB	1 GB	SCALE
no instances				
task	0	256 MB	1 GB	SCALE
no instances				

You can also review the app **Settings**. Specifically, we are looking at the configuration of the app once bound to the "apacheGeodeService" as seen in the **VCAP_SERVICES Environment Variable**:

The screenshot shows the Pivotal Apps Manager web interface. On the left is a dark sidebar with navigation links: Home, Marketplace, Accounting Report, Docs, and Tools. The main content area is titled 'Environment Variables' with a subtitle 'Defined by the runtime and buildpack. [Learn more](#)'. It displays a JSON configuration for environment variables. The JSON defines staging, running, and system environment variables, with a focus on VCAP_SERVICES for a 'user-provided' service named 'apacheGeodeService'. This service is configured with specific tags (gemfire, cloudcache, database, pivotal), instance name, binding name, credentials (including a locator and URL), and a user with 'admin' password and 'cluster_operator' role.

```
{
  "staging_env_json": {},
  "running_env_json": {},
  "system_env_json": {
    "VCAP_SERVICES": {
      "user-provided": [
        {
          "label": "user-provided",
          "name": "apacheGeodeService",
          "tags": [
            "gemfire",
            "cloudcache",
            "database",
            "pivotal"
          ],
          "instance_name": "apacheGeodeService",
          "binding_name": null,
          "credentials": {
            "locators": [
              "10.99.199.24[10334]"
            ],
            "urls": {
              "gfsh": "https://10.99.199.24:8080/gemfire/v1"
            },
            "users": [
              {
                "password": "admin",
                "roles": [
                  "cluster_operator"
                ],
                "username": "admin"
              }
            ]
          }
        }
      ]
    }
  }
}
```

This JSON document structure is not unlike the configuration used to bind your Spring Boot, **ClientCache** application to the Pivotal Cloud Cache service when deploying the same app to Pivotal CloudFoundry. This is actually very key if you want to minimize the amount of boilerplate code and configuration changes when migrating between different CloudFoundry environments, even [Open Source CloudFoundry](#).

Again, SBDG's entire goal is to simply the effort for you, as a developer, to build, run and manage your application, in whatever context your application lands, even if it changes later. If you follow the steps in this documentation, that goal will be realized.

22.4.5. Running the Spring Boot application

All that is left to do now is run the app.

You can start the PCCDemo app from the **cf** CLI using the following command:

Start the Spring Boot app

```
$ cf start boot-pcc-demo
```

Alternatively, you can also start the app from Apps Manager. This is convenient since then you can tail and monitor the application log file.

Once the app has started, you can click the [VIEW APP](#) link in the upper right corner of the **APP** screen.



191

You can also access the same data from the *Gfsh* command-line tool. However, the first thing to observe is that our application informed the cluster that it needed a Region called "Books":

Books Region

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

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

Non-Default Attributes Shared By Hosting Members

  Type |      Name      | Value
-----|-----|-----
Region| size           | 1
      | data-policy    | PARTITION
```

The PCCDemo app creates fake data on startup, which we can query in *Gfsh* like so:

Query Books

```
gfsh>query --query="SELECT book.isbn, book.title FROM /Books book"
Result : true
Limit  : 100
Rows   : 1

  isbn      | title
-----|-----
1235432BMF342 | The Torment of Others
```

22.5. Summary

There you have it!

The ability to deploy Spring Boot, Apache Geode **ClientCache** applications to Pivotal CloudFoundry, yet connect your app to an externally managed, standalone Apache Geode cluster is powerful.

Indeed, this will be a useful arrangement and stepping stone for many users as they begin their journey towards Cloud-Native platforms like Pivotal CloudFoundry and using services like Pivotal Cloud Cache.

Later, when the time comes and your need is real, you can simply migrate your Spring Boot

applications to a fully managed and production-grade Pivotal CloudFoundry environment and SBDG will figure out what to do, leaving you to focus entirely on your application.

Chapter 23. Docker

The state of modern software application development is moving towards [containerization](#). Containers offer a controlled environment to predictably build (configure & package), run and manage your applications in a reliable and repeatable manner regardless of context. The intrinsic benefit of using Containers is a no brainer.

Understandably, [Docker's](#) popularity took off like wildfire given its highly powerful and simplified model for creating, using and managing Containers to run packaged applications.

Docker's ecosystem is also quite impressive, with the event of [Testcontainers](#) along with Spring Boot's now [dedicated support](#) to create packaged Spring Boot apps in [Docker Images](#) that are then later run in a Docker Container.



Also see [Deploying to Containers](#) to learn more.

Apache Geode is no exception to being able to run in a controlled, containerized environment. The goal of this chapter is to get you started running Apache Geode in a Container and interfacing to a containerized Apache Geode cluster from your Spring Boot, Apache Geode client applications.

This chapter does not cover how to run your Spring Boot, Apache Geode client applications in a Container since that is already covered by Spring Boot (again, see [here](#) and [here](#), along with Docker's [docs](#)). Instead, our focus is on how to run an Apache Geode cluster in a Container and connect to it from a Spring Boot, Apache Geode client application, regardless of whether the app is running in a Container or not.

Let's get started.

23.1. Acquiring the Apache Geode Docker Image

To run an Apache Geode cluster inside a Docker Container you must first acquire the Docker Image. The Apache Geode Docker Image can be acquired from [Docker Hub](#).

While Apache Geode's official [documentation](#) is less than clear on how to use Apache Geode in Docker, we find a bit of relief in the [Wiki](#). However, for a complete and comprehensive write up, please refer to the instructions in the [README](#) from this [GitHub Repo](#).



You must have [Docker](#) installed on your local system to complete the following steps.

Effectively, the high-level steps are as follows:

- 1) Acquire the Apache Geode Docker Image from Docker Hub using the `docker pull` command from the command-line:

Download/Install the Apache Geode Docker Image

```
$ docker pull apachegeode/geode
Using default tag: latest
latest: Pulling from apachegeode/geode
Digest: sha256:6a6218f22a2895bb706175727c7d76f654f9162acac22b2d950d09a2649f9cf4
Status: Image is up to date for apachegeode/geode:latest
docker.io/apachegeode/geode:latest
```

Instead of pulling from the **nightly** TAG as suggested, the Spring team highly recommends that you pull from the **latest** TAG, which pulls a stable, production-ready Apache Geode Docker Image based on the latest Apache Geode GA version.

2) Verify the Apache Geode Docker Image was downloaded and installed successfully:

```
$ docker image ls
```

REPOSITORY	SIZE	TAG	IMAGE ID	
apachegeode/geode		latest	a2e210950712	2
months ago	224MB			
cloudfoundry/run		base-cnb	3a7d172559c2	8
weeks ago	71.2MB			
open-liberty		19.0.0.9-webProfile8	dece75feff1a	3
months ago	364MB			
tomee		11-jre-8.0.0-M3-webprofile	0d03e4d395e6	3
months ago	678MB			
...				

Now, you are ready to run Apache Geode in a Docker Container.

23.2. Running Apache Geode in a Docker Container

Now that we have acquired the Apache Geode Docker Image, we can run Apache Geode in a Docker Container. Use the following **docker run** command to start Apache Geode in a Docker Container:

Start the Apache Geode Docker Container

```
$ docker run -it -p 10334:10334 -p 40404:40404 -p 1099:1099 -p 7070:7070 -p 7575:7575
apachegeode/geode
```

```

  _____
 /  _  _/  _  _/  _  _/  _  _/
/  /  _/  /  _/  _  _/  _  _/
/  /  _/  _  _/  _  _/  _  _/
/  _  _/  _  _/  _  _/  _  _/  1.12.0
```

```
Monitor and Manage Apache Geode
gfsh>
```

Since the Apache Geode Docker Container was started in interactive mode, you must open a separate command-line shell to verify the Apache Geode Docker Container is in fact running:

Verify the Apache Geode Docker Container is Running

```
$ docker container ls
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS
PORTS
NAMES
3b30b9ffc5dc       apachegeode/geode  "gfsh"             44 seconds ago     Up 43
seconds            0.0.0.0:1099->1099/tcp, 0.0.0.0:7070->7070/tcp, 0.0.0.0:7575->7575/tcp,
0.0.0.0:10334->10334/tcp, 0.0.0.0:40404->40404/tcp, 8080/tcp    awesome_khorana
```

Of course, we know that the Apache Geode Docker Container is running since we ended up at a *Gfsh* command prompt in the interactive shell.

We also mapped ports between the Docker Container and the host system, exposing well-known ports used by Apache Geode server-side, cluster processes, such as Locators and Cache Servers.

Table 21. Apache Geode Ports

Process	Port
HTTP	7070
Locator	10334
Manager	1099
Server	40404

It is unfortunate that the Apache Geode Docker Image only gives you a *Gfsh* command prompt, leaving you with the task of provisioning a cluster. It would have been more useful to provide preconfigured Docker Images with different Apache Geode cluster configurations, such as 1 Locator + 1 Server, or 2 Locators + 4 Servers, etc. But, no matter, we can start the cluster ourselves.

23.3. Start an Apache Geode Cluster in Docker

From inside the Apache Geode Docker Container we can start a Locator and a Server.

Start Apache Geode Locator & Server

```
gfsh>start locator --name=LocatorOne --log-level=config --hostname-for
-clients=localhost
Starting a Geode Locator in /LocatorOne...
.....
Locator in /LocatorOne on 3b30b9ffc5dc[10334] as LocatorOne is currently online.
Process ID: 167
Uptime: 9 seconds
Geode Version: 1.12.0
Java Version: 1.8.0_212
Log File: /LocatorOne/LocatorOne.log
```

```
JVM Arguments: -Dgemfire.enable-cluster-configuration=true -Dgemfire.load-cluster
-configuration-from-dir=false -Dgemfire.log-level=config
-Dgemfire.launcher.registerSignalHandlers=true -Djava.awt.headless=true
-Dsun.rmi.dgc.server.gcInterval=9223372036854775806
Class-Path: /geode/lib/geode-core-1.12.0.jar:/geode/lib/geode-dependencies.jar
```

Successfully connected to: JMX Manager [host=3b30b9ffc5dc, port=1099]

Cluster configuration service is up and running.

```
gfsh>start server --name=ServerOne --log-level=config --hostname-for-clients=localhost
Starting a Geode Server in /ServerOne...
```

.....

Server in /ServerOne on 3b30b9ffc5dc[40404] as ServerOne is currently online.

Process ID: 267

Uptime: 7 seconds

Geode Version: 1.12.0

Java Version: 1.8.0_212

Log File: /ServerOne/ServerOne.log

JVM Arguments: -Dgemfire.default.locators=172.17.0.2[10334] -Dgemfire.start-dev-rest

-api=false -Dgemfire.use-cluster-configuration=true -Dgemfire.log-level=config

-Dgemfire.launcher.registerSignalHandlers=true -Djava.awt.headless=true

-Dsun.rmi.dgc.server.gcInterval=9223372036854775806

Class-Path: /geode/lib/geode-core-1.12.0.jar:/geode/lib/geode-dependencies.jar

```
gfsh>list members
```

Member Count : 2

Name		Id
LocatorOne		172.17.0.2(LocatorOne:167:locator)<ec><v0>:41000 [Coordinator]
ServerOne		172.17.0.2(ServerOne:267)<v1>:41001

```
gfsh>describe member --name=LocatorOne
```

Name : LocatorOne

Id : 172.17.0.2(LocatorOne:167:locator)<ec><v0>:41000

Host : 3b30b9ffc5dc

Regions :

PID : 167

Groups :

Used Heap : 50M

Max Heap : 443M

Working Dir : /LocatorOne

Log file : /LocatorOne/LocatorOne.log

Locators : 172.17.0.2[10334]

```
gfsh>describe member --name=ServerOne
```

```
Name      : ServerOne
Id        : 172.17.0.2(ServerOne:267)<v1>:41001
Host      : 3b30b9ffc5dc
Regions   :
PID       : 267
Groups    :
Used Heap : 77M
Max Heap  : 443M
Working Dir : /ServerOne
Log file  : /ServerOne/ServerOne.log
Locators  : 172.17.0.2[10334]
```

```
Cache Server Information
Server Bind      :
Server Port     : 40404
Running         : true
```

```
Client Connections : 0
```

We now have an Apache Geode cluster running with 1 Locator and 1 Server inside a Docker Container. We deliberately started the cluster with a minimal configuration. For example, we have no Regions in which to store data:

```
gfsh>list regions
No Regions Found
```

But, that is OK. Once more, we want to showcase the full power of SBDG and let the Spring Boot application drive the configuration of the Apache Geode cluster running in the Docker Container as required by the application.

Let's have a quick look at our Spring Boot application.

23.4. Spring Boot, Apache Geode client application explained

The Spring Boot, Apache Geode **ClientCache** application we will use to connect to our Apache Geode cluster running in the Docker Container appears as follows:

Spring Boot, Apache Geode Docker client application

```
@SpringBootApplication
@EnableClusterAware
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@UseMemberName("SpringBootApacheGeodeDockerClientCacheApplication")
public class SpringBootApacheGeodeDockerClientCacheApplication {

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



```

args);
    }

    @Bean
    @SuppressWarnings("unused")
    ApplicationRunner runner(GemFireCache cache, CustomerRepository
customerRepository) {

        return args -> {

            assertClientCacheAndConfigureMappingPdxSerializer(cache);
            assertThat(customerRepository.count()).isEqualTo(0);

            Customer jonDoe = Customer.newCustomer(1L, "Jon Doe");

            log("Saving Customer [%s]...%n", jonDoe);

            jonDoe = customerRepository.save(jonDoe);

            assertThat(jonDoe).isNotNull();
            assertThat(jonDoe.getId()).isEqualTo(1L);
            assertThat(jonDoe.getName()).isEqualTo("Jon Doe");
            assertThat(customerRepository.count()).isEqualTo(1);

            log("Querying for Customer [SELECT * FROM /Customers WHERE name LIKE
's']...%n", "%Doe");

            Customer queriedJonDoe = customerRepository.findByNameLike("%Doe");

            assertThat(queriedJonDoe).isEqualTo(jonDoe);

            log("Customer was [%s]%n", queriedJonDoe);
        };
    }

    private void assertClientCacheAndConfigureMappingPdxSerializer(GemFireCache cache)
{

        assertThat(cache).isNotNull();
        assertThat(cache.getName())

.isEqualTo(SpringBootApacheGeodeDockerClientCacheApplication.class.getSimpleName());
        assertThat(cache.getPdxSerializer()).assertInstanceOf(MappingPdxSerializer.class);

        MappingPdxSerializer serializer = (MappingPdxSerializer)
cache.getPdxSerializer();

        serializer.setIncludeTypeFilters(type -> Optional.ofNullable(type)
            .map(Class::getPackage)
            .map(Package::getName)
            .filter(packageName ->

```

```

packageName.startsWith(this.getClass().getPackage().getName()))
    .isPresent());
}

private void log(String message, Object... args) {
    System.err.printf(message, args);
    System.err.flush();
}
}

```

Our **Customer** application domain model object type is defined as:

Customer class

```

@Region("Customers")
class Customer {

    @Id
    private Long id;

    private String name;
}

```

And, we define a Spring Data CRUD *Repository* to persist and access **Customers** stored in the `"/Customers"` Region:

CustomerRepository interface

```

interface CustomerRepository extends CrudRepository<Customer, Long> {

    Customer findByNameLike(String name);
}

```

Our main class is annotated with **@SpringBootApplication** making it a proper Spring Boot application.

We additionally annotate the main class with SBDG's **@EnableClusterAware** to automatically detect the Apache Geode cluster running in the Docker Container as well as to push cluster configuration metadata from the application to the cluster as required by the application.

Specifically, the application requires that a Region called "Customers", as defined by the **@Region** mapping annotation on the **Customer** application domain model class, exists on the server(s) in the cluster to persist **Customer** data.

We use the SDG **@EnableEntityDefinedRegions** annotation to define the matching, client **PROXY** "Customers" Region.

Optionally, we have also annotated our main class with SBDG's `@UseMemberName` annotation to give the `ClientCache` a name, which we assert in the `assertClientCacheAndConfigurePdxSerializer(:ClientCache)` method.

The primary work performed by this application is done in the Spring Boot `ApplicationRunner` bean definition. We essentially create a `Customer` instance, "Jon Doe", save "Jon Doe" to the "Customers" Region managed by the server(s) in the cluster, and then query for "Jon Doe" using OQL, asserting that the result is equal to the expected.

We log the output from the application's operations to see the application in action.

23.5. Running the Spring Boot, Apache Geode client application

When you run the Spring Boot, Apache Geode client application, you should see output similar to:

Application log output

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java ...

org.springframework.geode.docs.example.app.docker.SpringBootApacheGeodeDockerClientCac
heApplication

      .
     /\ / ____' _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
    ( ( )\___| ' _ | ' _ | ' _ \ _ ' | \ \ \ \ \
   \ \ ___| |_) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
    ' |___| . _ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
   =====|_|=====|___/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_/_
  :: Spring Boot ::      (v2.3.0.RELEASE)

Saving Customer [Customer(name=Jon Doe)]...
Querying for Customer [SELECT * FROM /Customers WHERE name LIKE '%Doe']...
Customer was [Customer(name=Jon Doe)]

Process finished with exit code 0
```

Now when we review the configuration of the cluster, we see that the `/Customers` Region was created once the application has run:

```
gfsh>list regions
List of regions
-----
Customers

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

Non-Default Attributes Shared By Hosting Members

  Type |      Name      | Value
-----|-----|-----
Region | size           | 1
      | data-policy    | PARTITION
```

Our "/Customers" Region contains a value, "Jon Doe", and we can verify this by running the following OQL Query with *Gfsh*:

Query the "/Customers" Region

```
gfsh>query --query="SELECT customer.name FROM /Customers customer"
Result : true
Limit  : 100
Rows   : 1

Result
-----
Jon Doe
```

Indeed, our application ran successfully!

23.6. Conclusion

In this chapter, we saw how to connect a Spring Boot, Apache Geode **ClientCache** application to an Apache Geode cluster running in a Docker Container.

Later, we will provide more information on how to scale up, or rather scale out, our Apache Geode cluster running in Docker. Additionally, we will provide details on how you can use Apache Geode's Docker Image with **Testcontainers** when writing *Integration Tests*, which will formally become part of the Spring Test for Apache Geode (STDG) project.

Chapter 24. Samples

This section contains working examples demonstrating how to use Spring Boot for Apache Geode (SBDG) effectively.

Some examples focus on specific Use Cases (e.g. [(HTTP) Session state] caching) while other examples demonstrate how SBDG works under-the-hood to give users a better understanding of what is actually happening and how to debug problems with their Apache Geode, Spring Boot applications.

Table 22. Example Spring Boot applications using Apache Geode

Guide	Description	Source
Getting Started with Spring Boot for Apache Geode	Explains how to get started quickly, easily and reliably building Apache Geode and Pivotal Cloud Cache powered applications with Spring Boot.	Getting Started
Spring Boot Auto-Configuration for Apache Geode	Explains what auto-configuration is provided by SBDG out-of-the-box and what the auto-configuration is doing.	Boot Auto-Configuration
Spring Boot Actuator for Apache Geode	Explains how to use Spring Boot Actuator for Apache Geode and how it works.	Boot Actuator
Spring Boot Security for Apache Geode	Explains how to configure Auth and TLS with SSL when using Apache Geode and Pivotal Cloud Cache in Spring Boot applications.	Boot Security
Look-Aside Caching with Spring's Cache Abstraction and Apache Geode	Explains how to enable and use the Spring Cache Abstraction with Apache Geode as the caching provider for Look-Aside Caching.	Look-Aside Caching
Inline Caching with Spring's Cache Abstraction and Apache Geode	Explains how to enable and use the Spring Cache Abstraction with Apache Geode as the caching provider for Inline Caching. This sample builds on the <i>Look-Aside Caching</i> sample above.	Inline Caching

Guide	Description	Source
Asynchronous Inline Caching with Spring's Cache Abstraction and Apache Geode	Explains how to enable and use the Spring Cache Abstraction with Apache Geode as the caching provider for Asynchronous Inline Caching. This sample builds on the <i>Look-Aside Caching</i> and <i>Inline Caching</i> samples above.	Asynchronous Inline Caching
Near Caching with Spring's Cache Abstraction and Apache Geode	Explains how to enable and use the Spring Cache Abstraction with Apache Geode as the caching provider for Near Caching. This sample builds on the <i>Look-Aside Caching</i> sample above	Near Caching
Multi-Site Caching with Spring's Cache Abstraction and Apache Geode	Explains how to enable and use the Spring Cache Abstraction with Apache Geode as the caching provider for Multi-Site Caching. This sample builds on the <i>Look-Aside Caching</i> sample above and is the 4th and final leg in our study of <i>caching patterns</i> .	Multi-Site Caching
HTTP Session Caching with Spring Session and Apache Geode	Explains how to enable and use Spring Session with Apache Geode to manage HTTP Session state.	HTTP Session Caching

Chapter 25. Appendix

The following appendices provide additional help while developing Spring Boot applications backed by Apache Geode.

Table of Contents

1. [Auto-configuration vs. Annotation-based configuration](#)
2. [Configuration Metadata Reference](#)
3. [Disabling Auto-configuration](#)
4. [Switching from Apache Geode to Pivotal GemFire or Pivotal Cloud Cache \(PCC\)](#)
5. [Running an Apache Geode cluster using Spring Boot from your IDE](#)
6. [Testing](#)
7. [Examples](#)
8. [References](#)

Auto-configuration vs. Annotation-based configuration

The question most often asked is, "*What Spring Data for Apache Geode (SDG) annotations can I use, or must I use, when developing Apache Geode applications with Spring Boot?*"

This section will answer this question and more.

Readers should refer to the complimentary sample, [Spring Boot Auto-configuration for Apache Geode](#), which showcases the *auto-configuration* provided by Spring Boot for Apache Geode in action.

Background

To help answer this question, we must start by reviewing the complete collection of available Spring Data for Apache Geode (SDG) annotations. These annotations are provided in the [org.springframework.data.gemfire.config.annotation](#) package. Most of the pertinent annotations begin with `@Enable...`, except for the base annotations: `@ClientCacheApplication`, `@PeerCacheApplication` and `@CacheServerApplication`.

By extension, Spring Boot for Apache Geode (SBDG) builds on SDG's Annotation-based configuration model to implement *auto-configuration* and apply Spring Boot's core concepts, like "*convention over configuration*", enabling Apache Geode applications to be built with Spring Boot reliably, quickly and easily.

SDG provides this Annotation-based configuration model to, first and foremost, give application developers "*choice*" when building Spring applications using Apache Geode. SDG makes no assumptions about what application developers are trying to do and fails fast anytime the configuration is ambiguous, giving users immediate feedback.

Second, SDG's Annotations were meant to get application developers up and running quickly and reliably with ease. SDG accomplishes this by applying sensible defaults so application developers do not need to know, or even have to learn, all the intricate configuration details and tooling provided by Apache Geode to accomplish simple tasks, e.g. build a prototype.

So, SDG is all about "choice" and SBDG is all about "convention". Together these frameworks provide application developers with convenience and reliability to move quickly and easily.

To learn more about the motivation behind SDG's Annotation-based configuration model, refer to the [Reference Documentation](#).

Conventions

Currently, SBDG provides *auto-configuration* for the following features:

- `ClientCache`
- Caching with Spring's Cache Abstraction
- Continuous Query
- Function Execution & Implementation
- Logging
- PDX
- `GemfireTemplate`
- Spring Data Repositories
- Security (Client/Server Auth & SSL)
- Spring Session

Technically, this means the following SDG Annotations are not required to use the features above:

- `@ClientCacheApplication`
- `@EnableGemfireCaching` (or by using Spring Framework's `@EnableCaching`)
- `@EnableContinuousQueries`
- `@EnableGemfireFunctionExecutions`
- `@EnableGemfireFunctions`
- `@EnableLogging`
- `@EnablePdx`
- `@EnableGemfireRepositories`
- `@EnableSecurity`
- `@EnableSsl`
- `@EnableGemFireHttpSession`

Since SBDG auto-configures these features for you, then the above annotations are not strictly required. Typically, you would only declare one of these annotations when you want to "override"

Spring Boot's conventions, expressed in *auto-configuration*, and "customize" the behavior of the feature.

Overriding

In this section, we cover a few examples to make the behavior when overriding more apparent.

Caches

By default, SBDG provides you with a `ClientCache` instance. Technically, SBDG accomplishes this by annotating an auto-configuration class with `@ClientCacheApplication`, internally.

It is by convention that we assume most application developers' will be developing Spring Boot applications using Apache Geode as "client" applications in Apache Geode's client/server topology. This is especially true as users migrate their applications to a managed cloud environment.

Still, users are free to "override" the default settings and declare their Spring applications to be actual peer `Cache` members of a cluster, instead.

For example:

```
@SpringBootApplication
@CacheServerApplication
class MySpringBootPeerCacheServerApplication { }
```

By declaring the `@CacheServerApplication` annotation, you effectively override the SBDG default. Therefore, SBDG will not provide a `ClientCache` instance because you have informed SBDG of exactly what you want, i.e. a peer `Cache` instance hosting an embedded `CacheServer` that allows client connections.

However, you then might ask, *"Well, how do I customize the ClientCache instance when developing client applications without explicitly declaring the @ClientCacheApplication annotation, then?"*

First, you are entirely allowed to "customize" the `ClientCache` instance by explicitly declaring the `@ClientCacheApplication` annotation in your Spring Boot application configuration, and set specific attributes as needed. However, you should be aware that by explicitly declaring this annotation, or any of the other auto-configured annotations by default, then you assume all the responsibility that comes with it since you have effectively overridden the auto-configuration. One example of this is Security, which we touch on more below.

The most ideal way to "customize" the configuration of any feature is by way of the well-known and documented [Properties](#), specified in Spring Boot `application.properties` (the "convention"), or by using a [Configurer](#).

See the [Reference Guide](#) for more details.

Security

Like the `@ClientCacheApplication` annotation, the `@EnableSecurity` annotation is not strictly required,

not unless you want to override and customize the defaults.

Outside a managed environment, the only Security configuration required is specifying a username and password. You do this using the well-known and document SDG username/password properties in Spring Boot `application.properties`, like so:

Required Security Properties in a Non-Managed Environment

```
spring.data.gemfire.security.username=MyUser  
spring.data.gemfire.security.password=Secret
```

You do not need to explicitly declare the `@EnableSecurity` annotation just to specify Security configuration (e.g. username/password).

Inside a managed environment, such as the VMware Tanzu Application Service (TAS) when using VMware Tanzu GemFire, SBDG is able to introspect the environment and configure Security (Auth) completely without the need to specify any configuration, usernames / passwords, or otherwise. This is due in part because PCF supplies the security details in the VCAP environment when the app is deployed to TAS and bound to services (e.g. VMware Tanzu GemFire).

So, in short, you do not need to explicitly declare the `@EnableSecurity` annotation (or the `@ClientCacheApplication` for that matter).

However, if you do explicitly declare either the `@ClientCacheApplication` and/or `@EnableSecurity` annotations, guess what, you are now responsible for this configuration and SBDG's *auto-configuration* no longer applies.

While explicitly declaring `@EnableSecurity` makes more sense when "overriding" the SBDG Security *auto-configuration*, explicitly declaring the `@ClientCacheApplication` annotation most likely makes less sense with regard to its impact on Security configuration.

This is entirely due to the internals of Apache Geode, which in certain cases, like Security, not even Spring is able to completely shield users from the nuances of Apache Geode's configuration.

Both Auth and SSL must be configured before the cache instance (whether a `ClientCache` or a peer `Cache`, it does not matter) is created. Technically, this is because Security is enabled/configured during the "construction" of the cache. And, the cache pulls the configuration from JVM System properties that must be set before the cache is constructed.

Structuring the "exact" order of the *auto-configuration* classes provided by SBDG when the classes are triggered, is no small feat. Therefore, it should come as no surprise to learn that the Security *auto-configuration* classes in SBDG must be triggered before the `ClientCache` *auto-configuration* class, which is why a `ClientCache` instance cannot "auto" authenticate properly in PCC when the `@ClientCacheApplication` is explicitly declared without some assistance (i.e. you must also explicitly declare the `@EnableSecurity` annotation in this case since you overrode the *auto-configuration* of the cache, and, well, implicitly Security as well).

Again, this is due to the way Security (Auth) and SSL metadata must be supplied to Apache Geode.

See the [Reference Guide](#) for more details.

Extension

Most of the time, many of the other auto-configured annotations for CQ, Functions, PDX, Repositories, and so on, do not need to ever be declared explicitly.

Many of these features are enabled automatically by having SBDG or other libraries (e.g. Spring Session) on the classpath, or are enabled based on other annotations applied to beans in the Spring `ApplicationContext`.

Let's review a few examples.

Caching

It is rarely, if ever, necessary to explicitly declare either the Spring Framework's `@EnableCaching`, or the SDG specific `@EnableGemfireCaching` annotation, in Spring configuration when using SBDG. SBDG automatically "enables" caching and configures the SDG `GemfireCacheManager` for you.

You simply only need to focus on which application service components are appropriate for caching:

Service Caching

```
@Service
class CustomerService {

    @Autowired
    private CustomerRepository customerRepository;

    @Cacheable("CustomersByName")
    public Customer findBy(String name) {
        return customerRepository.findByName(name);
    }
}
```

Of course, it is necessary to create Apache Geode Regions backing the caches declared in your application service components (e.g. "CustomersByName") using Spring's Caching Annotations (e.g. `@Cacheable`), or alternatively, JSR-107, JCache annotations (e.g. `@CacheResult`).

You can do that by defining each Region explicitly, or more conveniently, you can simply use:

Configuring Caches (Regions)

```
@SpringBootApplication
@EnableCachingDefinedRegions
class Application { }
```

`@EnableCachingDefinedRegions` is optional, provided for convenience, and complimentary to caching when used rather than necessary.

See the [Reference Guide](#) for more details.

Continuous Query

It is rarely, if ever, necessary to explicitly declare the SDG `@EnableContinuousQueries` annotation. Instead, you should be focused on defining your application queries and worrying less about the plumbing.

For example:

Defining Queries for CQ

```
@Component
public class TemperatureMonitor extends AbstractTemperatureEventPublisher {

    @ContinuousQuery(name = "BoilingTemperatureMonitor",
        query = "SELECT * FROM /TemperatureReadings WHERE temperature.measurement >= 212.0")
    public void boilingTemperatureReadings(CqEvent event) {
        publish(event, temperatureReading -> new BoilingTemperatureEvent(this, temperatureReading));
    }

    @ContinuousQuery(name = "FreezingTemperatureMonitor",
        query = "SELECT * FROM /TemperatureReadings WHERE temperature.measurement <= 32.0")
    public void freezingTemperatureReadings(CqEvent event) {
        publish(event, temperatureReading -> new FreezingTemperatureEvent(this, temperatureReading));
    }
}
```

Of course, Apache Geode CQ only applies to clients.

See the [Reference Guide](#) for more details.

Functions

It is rarely, if ever, necessary to explicitly declare either the `@EnableGemfireFunctionExecutions` or `@EnableGemfireFunctions` annotations. SBDG provides *auto-configuration* for both Function implementations and executions. You simply need to define the implementation:

Function Implementation

```
@Component
class GeodeFunctions {

    @GemfireFunction
    Object exampleFunction(Object arg) {
        // ...
    }
}
```

And then define the execution:

Function Execution

```
@OnRegion(region = "Example")
interface GeodeFunctionExecutions {

    Object exampleFunction(Object arg);
}
```

SBDG will automatically find, configure and register Function Implementations (POJOs) in Apache Geode as proper **Functions** as well as create Executions proxies for the Interfaces which can then be injected into application service components to invoke the registered **Functions** without needing to explicitly declare the enabling annotations. The application Function Implementations & Executions (Interfaces) should simply exist below the **@SpringBootApplication** annotated main class.

See the <<[geode-functions,Reference Guide]>> for more details.

PDX

It is rarely, if ever, necessary to explicitly declare the **@EnablePdx** annotation since SBDG *auto-configures* PDX by default. SBDG automatically configures the SDG **MappingPdxSerializer** as the default **PdxSerializer** as well.

It is easy to customize the PDX configuration by setting the appropriate **Properties** (search for "PDX") in Spring Boot **application.properties**.

See the [Reference Guide](#) for more details.

Spring Data Repositories

It is rarely, if ever, necessary to explicitly declare the **@EnableGemfireRepositories** annotation since SBDG *auto-configures* Spring Data (SD) *Repositories* by default.

You simply only need to define your Repositories and get cranking:

Customer's Repository

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

    Customer findByName(String name);

}
```

SBDG finds the *Repository* interfaces defined in your application, proxies them, and registers them as beans in the Spring **ApplicationContext**. The *Repositories* may be injected into other application service components.

It is sometimes convenient to use the **@EnableEntityDefinedRegions** along with SD *Repositories* to identify the entities used by your application and define the Regions used by the SD *Repository*

infrastructure to persist the entity's state. The `@EnableEntityDefinedRegions` annotation is optional, provided for convenience, and complimentary to the `@EnableGemfireRepositories` annotation.

See the [Reference Guide](#) for more details.

Explicit Configuration

Most of the other annotations provided in SDG are focused on particular application concerns, or enable certain Apache Geode features, rather than being a necessity.

A few examples include:

- `@EnableAutoRegionLookup`
- `@EnableBeanFactoryLocator`
- `@EnableCacheServer(s)`
- `@EnableCachingDefinedRegions`
- `@EnableClusterConfiguration`
- `@EnableClusterDefinedRegions`
- `@EnableCompression`
- `@EnableDiskStore(s)`
- `@EnableEntityDefinedRegions`
- `@EnableEviction`
- `@EnableExpiration`
- `@EnableGatewayReceiver`
- `@EnableGatewaySender(s)`
- `@EnableGemFireAsLastResource`
- `@EnableHttpService`
- `@EnableIndexing`
- `@EnableOffHeap`
- `@EnableLocator`
- `@EnableManager`
- `@EnableMemcachedServer`
- `@EnablePool(s)`
- `@EnableRedisServer`
- `@EnableStatistics`
- `@UseGemFireProperties`

None of these annotations are necessary and none are auto-configured by SBDG. They are simply at the application developers disposal if and when needed. This also means none of these annotations are in conflict with any SBDG *auto-configuration*.

Summary

In conclusion, it is important to understand where SDG ends and SBDG begins. It all begins with the *auto-configuration* provided by SBDG out-of-the-box.

If a feature is not covered by SBDG's *auto-configuration*, then you are responsible for enabling and configuring the feature appropriately, as needed by your application (e.g. `@EnableRedisServer`).

In other cases, you might also want to explicitly declare a complimentary annotation (e.g. `@EnableEntityDefinedRegions`) for convenience, since there is no convention or "opinion" provided by SBDG out-of-the-box.

In all remaining cases, it boils down to understanding how Apache Geode works under-the-hood. While we go to great lengths to shield users from as many details as possible, it is not feasible or practical to address all matters, e.g. cache creation and Security.

Hope this section provided some relief and clarity.

Configuration Metadata Reference

The following 2 reference sections cover documented and well-known properties recognized and processed by *Spring Data for Apache Geode* (SDG) as well as *Spring Session for Apache Geode* (SSDG).

These properties may be used in Spring Boot `application.properties` files, or as JVM System properties, to configure different aspects of or enable individual features of Apache Geode in a Spring application. When combined with the power of Spring Boot, magical things begin to happen.

Spring Data Based Properties

The following properties all have a `spring.data.gemfire.*` prefix. For example, to set the cache `copy-on-read` property, use `spring.data.gemfire.cache.copy-on-read` in Spring Boot `application.properties`.

Table 23. `spring.data.gemfire.*` properties

Name	Description	Default	From
name	Name of the Apache Geode.	SpringBasedCacheClientApplication	<code>ClientCacheApplication.name</code>
locators	Comma-delimited list of Locator endpoints formatted as: locator1[port1],... ,locatorN[portN].	[]	<code>PeerCacheApplication.locators</code>

Name	Description	Default	From
use-bean-factory-locator	Enable the SDG BeanFactoryLocator when mixing Spring config with Apache Geode native config (e.g. cache.xml) and you wish to configure Apache Geode objects declared in <code>cache.xml</code> with Spring.	false	ClientCacheApplication.useBeanFactoryLocator

Table 24. `spring.data.gemfire.*` GemFireCache properties

Name	Description	Default	From
cache.copy-on-read	Configure whether a copy of an object returned from <code>Region.get(key)</code> is made.	false	ClientCacheApplication.copyOnRead
cache.critical-heap-percentage	Percentage of heap at or above which the cache is considered in danger of becoming inoperable.		ClientCacheApplication.criticalHeapPercentage
cache.critical-off-heap-percentage	Percentage of off-heap at or above which the cache is considered in danger of becoming inoperable.		ClientCacheApplication.criticalOffHeapPercentage
cache.enable-auto-region-lookup	Configure whether to lookup Regions configured in Apache Geode native config and declare them as Spring beans.	false	EnableAutoRegionLookup.enable
cache.eviction-heap-percentage	Percentage of heap at or above which the eviction should begin on Regions configured for HeapLRU eviction.		ClientCacheApplication.evictionHeapPercentage

Name	Description	Default	From
cache.eviction-off-heap-percentage	Percentage of off-heap at or above which the eviction should begin on Regions configured for HeapLRU eviction.		ClientCacheApplication.evictionOffHeapPercentage
cache.log-level	Configure the log-level of an Apache Geode cache.	config	ClientCacheApplication.logLevel
cache.name	Alias for 'spring.data.gemfire.name'.	SpringBasedCacheClientApplication	ClientCacheApplication.name
cache.compression.bean-name	Name of a Spring bean implementing org.apache.geode.compression.Compressor.		EnableCompression.compressorBeanName
cache.compression.region-names	Comma-delimited list of Region names for which compression will be configured.	[]	EnableCompression.regionNames
cache.off-heap.memory-size	Determines the size of off-heap memory used by Apache Geode in megabytes (m) or gigabytes (g); for example 120g.		EnableOffHeap.memorySize
cache.off-heap.region-names	Comma-delimited list of Region names for which off-heap will be configured.	[]	EnableOffHeap.regionNames

Table 25. `spring.data.gemfire.* ClientCache` properties

Name	Description	Default	From
cache.client.durable-client-id	Used only for clients in a client/server installation. If set, this indicates that the client is durable and identifies the client. The ID is used by servers to reestablish any messaging that was interrupted by client downtime.		ClientCacheApplication.durableClientId
cache.client.durable-client-timeout	Used only for clients in a client/server installation. Number of seconds this client can remain disconnected from its server and have the server continue to accumulate durable events for it.	300	ClientCacheApplication.durableClientTimeout
cache.client.keep-alive	Configure whether the server should keep the durable client's queues alive for the timeout period.	false	ClientCacheApplication.keepAlive

Table 26. `spring.data.gemfire.* peer` Cache properties

Name	Description	Default	From
cache.peer.enable-auto-reconnect	Configure whether member (Locators & Servers) will attempt to reconnect and reinitialize the cache after it has been forced out of the cluster by a network partition event or has otherwise been shunned by other members.	false	PeerCacheApplication.enableAutoReconnect

Name	Description	Default	From
cache.peer.lock-lease	Configures the length, in seconds, of distributed lock leases obtained by this cache.	120	PeerCacheApplication.lockLease
cache.peer.lock-timeout	Configures the number of seconds a cache operation will wait to obtain a distributed lock lease.	60	PeerCacheApplication.lockTimeout
cache.peer.message-sync-interval	Configures the frequency (in seconds) at which a message will be sent by the primary cache-server to all the secondary cache-server nodes to remove the events which have already been dispatched from the queue.	1	PeerCacheApplication.messageSyncInterval
cache.peer.search-timeout	Configures the number of seconds a cache get operation can spend searching for a value.	300	PeerCacheApplication.searchTimeout
cache.peer.use-cluster-configuration	Configures whether this cache member node would pull its configuration metadata from the cluster-based Cluster Configuration Service.	false	PeerCacheApplication.useClusterConfiguration

Table 27. `spring.data.gemfire.*` *CacheServer* properties

Name	Description	Default	From
cache.server.auto-startup	Configures whether the <i>CacheServer</i> should be started automatically at runtime.	true	CacheServerApplication.autoStartup

Name	Description	Default	From
cache.server.bind-address	Configures the IP address or hostname that this cache server will listen on.		CacheServerApplication.bindAddress
cache.server.hostname-for-clients	Configures the IP address or hostname that server locators will tell clients that this cache server is listening on.		CacheServerApplication.hostNameForClients
cache.server.load-poll-interval	Configures the frequency in milliseconds to poll the load probe on this cache server.	5000	CacheServerApplication.loadPollInterval
cache.server.max-connections	Configures the maximum client connections allowed.	800	CacheServerApplication.maxConnections
cache.server.max-message-count	Configures the maximum number of messages that can be enqueued in a client queue.	230000	CacheServerApplication.maxMessageCount
cache.server.max-threads	Configures the maximum number of threads allowed in this cache server to service client requests.		CacheServerApplication.maxThreads
cache.server.max-time-between-pings	Configures the maximum amount of time between client pings.	60000	CacheServerApplication.maxTimeBetweenPings
cache.server.message-time-to-live	Configures the time (in seconds) after which a message in the client queue will expire.	180	CacheServerApplication.messageTimeToLive
cache.server.port	Configures the port on which this cache server listens for clients.	40404	CacheServerApplication.port

Name	Description	Default	From
cache.server.socket-buffer-size	Configures buffer size of the socket connection to this CacheServer.	32768	CacheServerApplication.socketBufferSize
cache.server.subscription-capacity	Configures the capacity of the client queue.	1	CacheServerApplication.subscriptionCapacity
cache.server.subscription-disk-store-name	Configures the name of the DiskStore for client subscription queue overflow.		CacheServerApplication.subscriptionDiskStoreName
cache.server.subscription-eviction-policy	Configures the eviction policy that is executed when capacity of the client subscription queue is reached.	none	CacheServerApplication.subscriptionEvictionPolicy
cache.server.tcp-no-delay	Configures the outgoing Socket connection tcp-no-delay setting.	true	CacheServerApplication.tcpNoDelay

CacheServer properties can be further targeted at specific *CacheServer* instances, using an option bean name of the **CacheServer** bean defined in the Spring application context. For example:

```
spring.data.gemfire.cache.server.[<cacheServerBeanName>].bind-address=...
```

Table 28. **spring.data.gemfire.* Cluster properties**

Name	Description	Default	From
cluster.region.type	Configuration setting used to specify the data management policy used when creating Regions on the servers in the cluster.	RegionShortcut.PARTITION	EnableClusterConfiguration.serverRegionShortcut

Table 29. **spring.data.gemfire.* DiskStore properties**

Name	Description	Default	From
disk.store.allow-force-compaction	Configures whether to allow <code>DiskStore.forceCompaction()</code> to be called on Regions using a <code>DiskStore</code> .	false	EnableDiskStore.allowForceCompaction
disk.store.auto-compact	Configures whether to cause the disk files to be automatically compacted.	true	EnableDiskStore.autoCompact
disk.store.compaction-threshold	Configures the threshold at which an oplog will become compactable.	50	EnableDiskStore.compactionThreshold
disk.store.directory.location	Configures the system directory where the <code>DiskStore</code> (oplog) files will be stored.	[]	EnableDiskStore.diskDirectories.location
disk.store.directory.size	Configures the amount of disk space allowed to store <code>DiskStore</code> (oplog) files.	21474883647	EnableDiskStore.diskDirectories.size
disk.store.disk-usage-critical-percentage	Configures the critical threshold for disk usage as a percentage of the total disk volume.	99.0	EnableDiskStore.diskUsageCriticalPercentage
disk.store.disk-usage-warning-percentage	Configures the warning threshold for disk usage as a percentage of the total disk volume.	90.0	EnableDiskStore.diskUsageWarningPercentage
disk.store.max-oplog-size	Configures the maximum size in megabytes a single oplog (operation log) is allowed to be.	1024	EnableDiskStore.maxOplogSize

Name	Description	Default	From
disk.store.queue-size	Configures the maximum number of operations that can be asynchronously queued.		EnableDiskStore.queueSize
disk.store.time-interval	Configures the number of milliseconds that can elapse before data written asynchronously is flushed to disk.	1000	EnableDiskStore.timeInterval
disk.store.write-buffer-size	Configures the write buffer size in bytes.	32768	EnableDiskStore.writeBufferSize

DiskStore properties can be further targeted at specific *DiskStores* using the `DiskStore.name`.

For instance, you may specify directory location of the files for a specific, named `DiskStore` using:

```
spring.data.gemfire.disk.store.Example.directory.location=/path/to/geode/disk-stores/Example/
```

The directory location and size of the *DiskStore* files can be further divided into multiple locations and size using array syntax, as in:

```
spring.data.gemfire.disk.store.Example.directory[0].location=/path/to/geode/disk-stores/Example/one
spring.data.gemfire.disk.store.Example.directory[0].size=4096000
spring.data.gemfire.disk.store.Example.directory[1].location=/path/to/geode/disk-stores/Example/two
spring.data.gemfire.disk.store.Example.directory[1].size=8192000
```

Both the name and array index are optional and you can use any combination of name and array index. Without a name, the properties apply to all *DiskStores*. Without array indexes, all [named] *DiskStore* files will be stored in the specified location and limited to the defined size.

Table 30. `spring.data.gemfire.*` Entity properties

Name	Description	Default	From
entities.base-packages	Comma-delimited list of package names indicating the start points for the entity scan.		EnableEntityDefinedRegions.basePackages

Table 31. `spring.data.gemfire.*` Locator properties

Name	Description	Default	From
locator.host	Configures the IP address or hostname of the system NIC to which the embedded Locator will be bound to listen for connections.		EnableLocator.host
locator.port	Configures the network port to which the embedded Locator will listen for connections.	10334	EnableLocator.port

Table 32. `spring.data.gemfire.*` Logging properties

Name	Description	Default	From
logging.level	Configures the log-level of an Apache Geode cache; Alias for 'spring.data.gemfire.cache.log-level'.	config	EnableLogging.logLevel
logging.log-disk-space-limit	Configures the amount of disk space allowed to store log files.		EnableLogging.logDiskSpaceLimit
logging.log-file	Configures the pathname of the log file used to log messages.		EnableLogging.logFile
logging.log-file-size	Configures the maximum size of a log file before the log file is rolled.		EnableLogging.logFileSize

Table 33. `spring.data.gemfire.*` Management properties

Name	Description	Default	From
management.use-http	Configures whether to use the HTTP protocol to communicate with a Apache Geode Manager.	false	EnableClusterConfiguration.useHttp
management.http.host	Configures the IP address or hostname of the Apache Geode Manager running the HTTP service.		EnableClusterConfiguration.host
management.http.port	Configures the port used by the Apache Geode Manager's HTTP service to listen for connections.	7070	EnableClusterConfiguration.port

Table 34. `spring.data.gemfire.*` Manager properties

Name	Description	Default	From
manager.access-file	Configures the Access Control List (ACL) file used by the Manager to restrict access to the JMX MBeans by the clients.		EnableManager.accessFile
manager.bind-address	Configures the IP address or hostname of the system NIC used by the Manager to bind and listen for JMX client connections.		EnableManager.bindAddress
manager.hostname-for-clients	Configures the hostname given to JMX clients to ask the Locator for the location of the Manager.		EnableManager.hostNameForClients

Name	Description	Default	From
manager.password-file	By default, the JMX Manager will allow clients without credentials to connect. If this property is set to the name of a file then only clients that connect with credentials that match an entry in this file will be allowed.		EnableManager.passwordFile
manager.port	Configures the port used by the Manager to listen for JMX client connections.	1099	EnableManager.port
manager.start	Configures whether to start the Manager service at runtime.	false	EnableManager.start
manager.update-rate	Configures the rate, in milliseconds, at which this member will push updates to any JMX Managers.	2000	EnableManager.updateRate

Table 35. `spring.data.gemfire.*` PDX properties

Name	Description	Default	From
pdx.disk-store-name	Configures the name of the DiskStore used to store PDX type meta-data to disk when PDX is persistent.		EnablePdx.diskStoreName
pdx.ignore-unread-fields	Configures whether PDX ignores fields that were unread during deserialization.	false	EnablePdx.ignoreUnreadFields
pdx.persistent	Configures whether PDX persists type meta-data to disk.	false	EnablePdx.persistent

Name	Description	Default	From
pdx.read-serialized	Configures whether a Region entry is returned as a PdxInstance or deserialized back into object form on read.	false	EnablePdx.readSerialized
pdx.serialize-bean-name	Configures the name of a custom Spring bean implementing org.apache.geode.pdx.PdxSerializer.		EnablePdx.serializerBeanName

Table 36. `spring.data.gemfire.*` Pool properties

Name	Description	Default	From
pool.free-connection-timeout	Configures the timeout used to acquire a free connection from a Pool.	10000	EnablePool.freeConnectionTimeout
pool.idle-timeout	Configures the amount of time a connection can be idle before expiring (and closing) the connection.	5000	EnablePool.idleTimeout
pool.load-conditioning-interval	Configures the interval for how frequently the pool will check to see if a connection to a given server should be moved to a different server to improve the load balance.	300000	EnablePool.loadConditioningInterval
pool.locators	Comma-delimited list of Locator endpoints in the format: locator1[port1],...,locatorN[portN]		EnablePool.locators

Name	Description	Default	From
pool.max-connections	Configures the maximum number of client to server connections that a Pool will create.		EnablePool.maxConnections
pool.min-connections	Configures the minimum number of client to server connections that a Pool will maintain.	1	EnablePool.minConnections
pool.multi-user-authentication	Configures whether the created Pool can be used by multiple authenticated users.	false	EnablePool.multiUserAuthentication
pool.ping-interval	Configures how often to ping servers to verify that they are still alive.	10000	EnablePool.pingInterval
pool.pr-single-hop-enabled	Configures whether to perform single-hop data access operations between the client and servers. When true the client is aware of the location of partitions on servers hosting Regions with DataPolicy.PARTITION.	true	EnablePool.prSingleHopEnabled
pool.read-timeout	Configures the number of milliseconds to wait for a response from a server before timing out the operation and trying another server (if any are available).	10000	EnablePool.readTimeout

Name	Description	Default	From
pool.ready-for-events	Configures whether to signal the server that the client is prepared and ready to receive events.	false	ClientCacheApplication.readyForEvents
pool.retry-attempts	Configures the number of times to retry a request after timeout/exception.		EnablePool.retryAttempts
pool.server-group	Configures the group that all servers a Pool connects to must belong to.		EnablePool.serverGroup
pool.servers	Comma-delimited list of CacheServer endpoints in the format: server1[port1],... ,serverN[portN]		EnablePool.servers
pool.socket-buffer-size	Configures the socket buffer size for each connection made in all Pools.	32768	EnablePool.socketBufferSize
pool.statistic-interval	Configures how often to send client statistics to the server.		EnablePool.statisticInterval
pool.subscription-ack-interval	Configures the interval in milliseconds to wait before sending acknowledgements to the CacheServer for events received from the server subscriptions.	100	EnablePool.subscriptionAckInterval
pool.subscription-enabled	Configures whether the created Pool will have server-to-client subscriptions enabled.	false	EnablePool.subscriptionEnabled

Name	Description	Default	From
pool.subscription-message-tracking-timeout	Configures the messageTrackingTimeout attribute which is the time-to-live period, in milliseconds, for subscription events the client has received from the server.	900000	EnablePool.subscriptionMessageTrackingTimeout
pool.subscription-redundancy	Configures the redundancy level for all Pools server-to-client subscriptions.		EnablePool.subsriptionRedundancy
pool.thread-local-connections	Configures the thread local connections policy for all Pools.	false	EnablePool.threadLocalConnections

Table 37. `spring.data.gemfire.*` Security properties

Name	Description	Default	From
security.username	Configures the name of the user used to authenticate with the servers.		EnableSecurity.securityUsername
security.password	Configures the user password used to authenticate with the servers.		EnableSecurity.securityPassword
security.properties-file	Configures the system pathname to a properties file containing security credentials.		EnableAuth.propertiesFile
security.client.accessor	X	X	EnableAuth.clientAccessor

Name	Description	Default	From
security.client.accessor-post-processor	The callback that should be invoked in the post-operation phase, which is when the operation has completed on the server but before the result is sent to the client.		EnableAuth.clientAccessorPostProcessor
security.client.authentication-initializer	Static creation method returning an AuthInitialize object, which obtains credentials for peers in a cluster.		EnableSecurity.clientAuthenticationInitializer
security.client.authentication	Static creation method returning an Authenticator object used by a cluster member (Locator, Server) to verify the credentials of a connecting client.		EnableAuth.clientAuthenticator
security.client.diffie-hellman-algorithm	Used for authentication. For secure transmission of sensitive credentials like passwords, you can encrypt the credentials using the Diffie-Hellman key-exchange algorithm. Do this by setting the security-client-dhalgo system property on the clients to the name of a valid, symmetric key cipher supported by the JDK.		EnableAuth.clientDiffieHellmanAlgorithm

Name	Description	Default	From
security.log.file	Configures the pathname to a log file used for security log messages.		EnableAuth.securityLogFile
security.log.level	Configures the log-level for security log messages.		EnableAuth.securityLogLevel
security.manager.class-name	Configures name of a class implementing org.apache.geode.security.SecurityManager.		EnableSecurity.securityManagerClassName
security.peer.authentication-initializer	Static creation method returning an AuthInitialize object, which obtains credentials for peers in a cluster.		EnableSecurity.peerAuthenticationInitializer
security.peer.authenticator	Static creation method returning an Authenticator object, which is used by a peer to verify the credentials of a connecting node.		EnableAuth.peerAuthenticator
security.peer.verify-member-timeout	Configures the timeout in milliseconds used by a peer to verify membership of an unknown authenticated peer requesting a secure connection.		EnableAuth.peerVerifyMemberTimeout

Name	Description	Default	From
security.post-processor.class-name	Configures the name of a class implementing the org.apache.geode.security.PostProcessor interface that can be used to change the returned results of Region get operations.		EnableSecurity.securityPostProcessorClassName
security.shiro.ini-resource-path	Configures the Apache Geode System Property referring to the location of an Apache Shiro INI file that configures the Apache Shiro Security Framework in order to secure Apache Geode.		EnableSecurity.shiroIniResourcePath

Table 38. `spring.data.gemfire.*` SSL properties

Name	Description	Default	From
security.ssl.certificate.alias.cluster	Configures the alias to the stored SSL certificate used by the cluster to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.alias.default-alias	Configures the default alias to the stored SSL certificate used to secure communications across the entire Apache Geode system.		EnableSsl.defaultCertificateAlias
security.ssl.certificate.alias.gateway	Configures the alias to the stored SSL certificate used by the WAN Gateway Senders/Receivers to secure communications.		EnableSsl.componentCertificateAliases

Name	Description	Default	From
security.ssl.certificate.alias.jmx	Configures the alias to the stored SSL certificate used by the Manager's JMX based JVM MBeanServer and JMX clients to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.alias.locator	Configures the alias to the stored SSL certificate used by the Locator to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.alias.server	Configures the alias to the stored SSL certificate used by clients and servers to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.alias.web	Configures the alias to the stored SSL certificate used by the embedded HTTP server to secure communications (HTTPS).		EnableSsl.componentCertificateAliases
security.ssl.ciphers	Comma-separated list of SSL ciphers or "any".		EnableSsl.ciphers
security.ssl.components	Comma-delimited list of Apache Geode components (e.g. WAN) to be configured for SSL communication.		EnableSsl.components
security.ssl.keystore	Configures the system pathname to the Java KeyStore file storing certificates for SSL.		EnableSsl.keystore

Name	Description	Default	From
security.ssl.keystore.password	Configures the password used to access the Java KeyStore file.		EnableSsl.keystorePassword
security.ssl.keystore.type	Configures the password used to access the Java KeyStore file (e.g. JKS).		EnableSsl.keystoreType
security.ssl.protocols	Comma-separated list of SSL protocols or “any”.		EnableSsl.protocols
security.ssl.require-authentication	Configures whether 2-way authentication is required.		EnableSsl.requireAuthentication
security.ssl.truststore	Configures the system pathname to the trust store (Java KeyStore file) storing certificates for SSL.		EnableSsl.truststore
security.ssl.truststore.password	Configures the password used to access the trust store (Java KeyStore file).		EnableSsl.truststorePassword
security.ssl.truststore.type	Configures the password used to access the trust store (Java KeyStore file; e.g. JKS).		EnableSsl.truststoreType
security.ssl.web-require-authentication	Configures whether 2-way HTTP authentication is required.	false	EnableSsl.webRequireAuthentication

Table 39. `spring.data.gemfire.*` Service properties

Name	Description	Default	From
service.http.bind-address	Configures the IP address or hostname of the system NIC used by the embedded HTTP server to bind and listen for HTTP(S) connections.		EnableHttpService.bindAddress
service.http.port	Configures the port used by the embedded HTTP server to listen for HTTP(S) connections.	7070	EnableHttpService.port
service.http.ssl-require-authentication	Configures whether 2-way HTTP authentication is required.	false	EnableHttpService.sslRequireAuthentication
service.http.dev-rest-api-start	Configures whether to start the Developer REST API web service. A full installation of Apache Geode is required and you must set the \$GEODE environment variable.	false	EnableHttpService.startDeveloperRestApi
service.memcached.port	Configures the port of the embedded Memcached server (service).	11211	EnableMemcachedServer.port
service.memcached.protocol	Configures the protocol used by the embedded Memcached server (service).	ASCII	EnableMemcachedServer.protocol

Name	Description	Default	From
service.redis.bind-address	Configures the IP address or hostname of the system NIC used by the embedded Redis server to bind and listen for connections.		EnableRedis.bindAddress
service.redis.port	Configures the port used by the embedded Redis server to listen for connections.	6479	EnableRedisServer.port

Spring Session Based Properties

The following properties all have a `spring.session.data.gemfire.*` prefix. For example, to set the Session Region name, use `spring.session.data.gemfire.session.region.name` in `Spring Boot application.properties`.

Table 40. `spring.session.data.gemfire.*` properties

Name	Description	Default	From
cache.client.pool.name	Name of the Pool used to send data access operations between the client and server(s).	gemfirePool	EnableGemFireHttpSession.poolName
cache.client.region.shortcut	Configures the DataPolicy used by the client Region to manage (HTTP) Session state.	ClientRegionShortcut.PROXY	EnableGemFireHttpSession.clientRegionShortcut
cache.server.region.shortcut	Configures the DataPolicy used by the server Region to manage (HTTP) Session state.	RegionShortcut.PARTITION	EnableGemFireHttpSession.serverRegionShortcut
session.attributes.indexable	Configures names of Session attributes for which an Index will be created.	[]	EnableGemFireHttpSession.indexableSessionAttributes

Name	Description	Default	From
session.expiration.max-inactive-interval-seconds	Configures the number of seconds in which a Session can remain inactive before it expires.	1800	EnableGemFireHttpSession.maxInactiveIntervalSeconds
session.region.name	Configures name of the (client/server) Region used to manage (HTTP) Session state.	ClusteredSpringSessions	EnableGemFireHttpSession.regionName
session.serializer.bean-name	Configures the name of a Spring bean implementing <code>org.springframework.session.data.gemfire.serialization.SessionSerializer</code> .		EnableGemFireHttpSession.sessionSerializerBeanName

Apache Geode Properties

While it is not recommended to use Apache Geode properties directly in your Spring applications, SBDG will not prevent you from doing so. A complete reference to the Apache Geode specific properties can be found [here](#).



Apache Geode is very strict about the properties that maybe specified in a `gemfire.properties` file. You cannot mix Spring properties with `gemfire.*` properties in an Apache Geode `gemfire.properties` file.

Disabling Auto-configuration

If you would like to disable the *auto-configuration* of any feature provided by Spring Boot for Apache Geode, then you can specify the *auto-configuration* class in the `exclude` attribute of the `@SpringBootApplication` annotation, as follows:

Disable Auto-configuration of PDX

```
@SpringBootApplication(exclude = PdxSerializationAutoConfiguration.class)
public class MySpringBootApplication {

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

Of course, you can disable more than 1 *auto-configuration* class at a time by specifying each class in

the `exclude` attribute using array syntax, as follows:

Disable Auto-configuration of PDX & SSL

```
@SpringBootApplication(exclude = { PdxSerializationAutoConfiguration.class,  
    SslAutoConfiguration.class })  
public class MySpringBootApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(MySpringBootApplication.class, args);  
    }  
}
```

Complete Set of Auto-configuration Classes

The current set of *auto-configuration* classes in Spring Boot for Apache Geode include:

- `CacheNameAutoConfiguration`
- `CachingProviderAutoConfiguration`
- `ClientCacheAutoConfiguration`
- `ClientSecurityAutoConfiguration`
- `ContinuousQueryAutoConfiguration`
- `FunctionExecutionAutoConfiguration`
- `GemFirePropertiesAutoConfiguration`
- `LoggingAutoConfiguration`
- `PdxSerializationAutoConfiguration`
- `PeerSecurityAutoConfiguration`
- `RegionTemplateAutoConfiguration`
- `RepositoriesAutoConfiguration`
- `SpringSessionAutoConfiguration`
- `SpringSessionPropertiesAutoConfiguration`
- `SslAutoConfiguration`

Switching from Apache Geode to Pivotal GemFire or Pivotal Cloud Cache (PCC)



This section is now deprecated! Spring Boot for Apache Geode (SBDG) no longer provides the `spring-gemfire-starter` and related starter modules. As of SBDG 1.4, SBDG is based on Apache Geode 1.13. Standalone GemFire bits based on Apache Geode are no longer being released by VMware, Inc. after GemFire 9.10. GemFire 9.10 was based on Apache Geode 1.12, and as such, SBDG can no longer properly support standalone GemFire bits (i.e. \leq 9.10).



What was "*Pivotal GemFire*" has now been rebranded as [VMware Tanzu GemFire](#) and what was Pivotal Cloud Cache (PCC) running on Pivotal CloudFoundry (PCF) has been rebranded as [VMware Tanzu GemFire for VMs](#) and [VMware Tanzu Application Service \(TAS\)](#), respectively.

Running an Apache Geode cluster using Spring Boot from your IDE

As described in [Building ClientCache Applications](#), it is possible to configure and run a small Apache Geode cluster from inside your IDE using Spring Boot. This is extremely helpful during development since it allows you to manually spin up, test and debug your applications quickly and easily.

Spring Boot for Apache Geode includes such a class:


```
@SpringBootApplication
@CacheServerApplication(name = "SpringBootApacheGeodeCacheServerApplication")
@SuppressWarnings("unused")
public class SpringBootApacheGeodeCacheServerApplication {

    public static void main(String[] args) {

        new
SpringApplicationBuilder(SpringBootApacheGeodeCacheServerApplication.class)
        .web(WebApplicationType.NONE)
        .build()
        .run(args);
    }

    @Configuration
    @UseLocators
    @Profile("clustered")
    static class ClusteredConfiguration { }

    @Configuration
    @EnableLocator
    @EnableManager(start = true)
    @Profile("!clustered")
    static class LonerConfiguration { }

}
```

This class is a proper Spring Boot application that can be used to configure and bootstrap multiple Apache Geode servers and joining them together to form a small cluster simply by modifying the runtime configuration of this class ever so slightly.

Initially you will want to start a single, primary server with the embedded Locator and Manager service.

The Locator service enables members in the cluster to locate one another and allows new members to attempt to join the cluster as a peer. Additionally, the Locator service also allows clients to connect to the servers in the cluster. When the cache client's Pool is configured to use Locators, then the Pool can intelligently route data requests directly to the server hosting the data (a.k.a. single-hop access), especially when the data is partitioned/sharded across servers in the cluster. Locator Pools include support for load balancing connections and handling automatic fail-over in the event of failed connections, among other things.

The Manager service enables you to connect to this server using *Gfsh* (the Apache Geode [command-line shell tool](#)).

To start our primary server, create a run configuration in your IDE for the `SpringBootApacheGeodeCacheServerApplication` class with the following, recommended JRE command-line options:

```
-server -ea -Dspring.profiles.active=
```

Start the class. You should see similar output:

Server 1 output on startup

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/bin/java -server -ea
-Dspring.profiles.active= "-javaagent:/Applications/IntelliJ IDEA 17
CE.app/Contents/lib/idea_rt.jar=62866:/Applications/IntelliJ IDEA 17
CE.app/Contents/bin" -Dfile.encoding=UTF-8 -classpath
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/charsets.jar:
/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/deploy.jar:/L
ibrary/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/cldrdata.ja
r:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/dnsns.j
ar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/jacce
s.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/jfx
rt.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/lo
caledata.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib
/ext/nashorn.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/l
ib/ext/sunec.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/
lib/ext/sunjce_provider.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Conten
ts/Home/jre/lib/ext/sunpkcs11.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Co
ntents/Home/jre/lib/ext/zipfs.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/C
ontents/Home/jre/lib/javaws.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Con
tents/Home/jre/lib/jce.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents
/Home/jre/lib/jfr.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home
/jre/lib/jfxswt.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/j
re/lib/jsse.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/l
ib/management-
agent.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/plu
gin.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/resou
rces.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/rt.j
ar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/ant-
javafx.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/dt.jar
:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/javafx-
mx.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/jconsole.j
ar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/packager.jar:/
Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/sa-
jdi.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/lib/tools.jar
:/Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
docs/build/classes/main:/Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
docs/build/resources/main:/Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
autoconfigure/build/classes/main:/Users/jblum/pivdev/spring-boot-data-geode/spring-
geode-autoconfigure/build/resources/main:/Users/jblum/pivdev/spring-boot-data-
geode/spring-geode/build/classes/main:/Users/jblum/.gradle/caches/modules-2/files-
2.1/org.springframework.boot/spring-boot-
starter/2.0.3.RELEASE/ffaa050dbd36b0441645598f1a7ddaf67fd5e678/spring-boot-starter-
2.0.3.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
```

2.1/org.springframework.boot/spring-boot-
 autoconfigure/2.0.3.RELEASE/11bc4cc96b08fabad2b3186755818fa0b32d83f/spring-boot-
 autoconfigure-2.0.3.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework.boot/spring-
 boot/2.0.3.RELEASE/b874870d915adbc3dd932e19077d3d45c8e54aa0/spring-boot-
 2.0.3.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/javax.annotation/javax.annotation-
 api/1.3.2/934c04d3cfef185a8008e7bf34331b79730a9d43/javax.annotation-api-
 1.3.2.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework.data/spring-data-
 geode/2.0.8.RELEASE/9e0a3cd2805306d355c77537aea07c281fc581b/spring-data-geode-
 2.0.8.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-context-
 support/5.0.7.RELEASE/e8ee4902d9d8bfbb21bc5e8f30cfbb4324adb4f3/spring-context-support-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 context/5.0.7.RELEASE/243a23f8968de8754d8199d669780d683ab177bd/spring-context-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 tx/5.0.7.RELEASE/4ca59b21c61162adb146ad1b40c30b60d8dc42b8/spring-tx-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 web/5.0.7.RELEASE/2e04c6c2922fbfa06b5948be14a5782db168b6ec/spring-web-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework.data/spring-data-
 commons/2.0.8.RELEASE/5c19af63b5acb0eab39066684e813d5ecd9d03b7/spring-data-commons-
 2.0.8.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 aop/5.0.7.RELEASE/fdd0b6aa3c9c7a188c3bfbf6dfd8d40e843be9ef/spring-aop-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 beans/5.0.7.RELEASE/c1196cb3e56da83e3c3a02ef323699f4b05feedc/spring-beans-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 expression/5.0.7.RELEASE/ca01fb473f53dd0ee3c85663b26d5dc325602057/spring-expression-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.springframework/spring-
 core/5.0.7.RELEASE/54b731178d81e66eca9623df772ff32718208137/spring-core-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.yaml/snakeyaml/1.19/2d998d3d674b172a588e54ab619854d073f555b5/snakeyaml-
 1.19.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.springframework/spring-
 jcl/5.0.7.RELEASE/699016ddf454c2c167d9f84ae5777eccadf54728/spring-jcl-
 5.0.7.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.apache.geode/geode-
 lucene/1.2.1/3d22a050bd4eb64bd8c82a74677f45c070f102d5/geode-lucene-
 1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.geode/geode-
 core/1.2.1/fe853317e33dd2a1c291f29cee3c4be549f75a69/geode-core-
 1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.geode/geode-
 cq/1.2.1/69873d6b956ba13b55c894a13e72106fb552e840/geode-cq-
 1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.geode/geode-
 wan/1.2.1/df0dd8516e1af17790185255ff21a54b56d94344/geode-wan-

1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/antlr/antlr/2.7.7/83cd2cd674a217ade95a4bb83a8a14f351f48bd0/antlr-
 2.7.7.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.shiro/shiro-
 spring/1.3.2/281a6b565f6cf3aebd31ddb004632008d7106f2d/shiro-spring-
 1.3.2.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.aspectj/aspectjweaver/1.8.13/ad94df2a28d658a40dc27bbaff6a1ce5fbf04e9b/aspectjw-
 eaver-1.8.13.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/com.fasterxml.jackson.core/jackson-
 databind/2.9.6/cfa4f316351a91bfd95cb0644c6a2c95f52db1fc/jackson-databind-
 2.9.6.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/com.fasterxml.jackson.core/jackson-
 annotations/2.9.0/7c10d545325e3a6e72e06381afe469fd40eb701/jackson-annotations-
 2.9.0.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.shiro/shiro-
 web/1.3.2/725be023e1c65a0fd70c01b8c0c13a2936c23315/shiro-web-
 1.3.2.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.shiro/shiro-
 core/1.3.2/b5dede9d890f335998a8ebf479809fe365b927fc/shiro-core-
 1.3.2.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.slf4j/slf4j-
 api/1.7.25/da76ca59f6a57ee3102f8f9bd9cee742973efa8a/slf4j-api-
 1.7.25.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/com.github.stephenc.findbugs/findbugs-annotations/1.3.9-
 1/a6b11447635d80757d64b355bed3c00786d86801/findbugs-annotations-1.3.9-
 1.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.jgroups/jgroups/3.6.10.Final/fc0ff5a8a9de27ab62939956f705c2909bf86bc2/jgroups-
 3.6.10.Final.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/commons-io/commons-
 io/2.5/2852e6e05fbb95076fc091f6d1780f1f8fe35e0f/commons-io-
 2.5.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/commons-lang/commons-
 lang/2.6/ce1edb914c94ebc388f086c6827e8bdeec71ac2/commons-lang-
 2.6.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/it.unimi.dsi/fastutil/7.1.0/9835253257524c1be7ab50c057aa2d418fb72082/fastutil-
 7.1.0.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/javax.resource/javax.resource-
 api/1.7/ae40e0864eb1e92c48bf82a2a3399cbbf523fb79/javax.resource-api-
 1.7.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/net.java.dev.jna/jna/4.5.1/65bd0cacc9c79a21c6ed8e9f588577cd3c2f85b9/jna-
 4.5.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/net.sf.jopt-simple/jopt-
 simple/5.0.3/cdd846cfc4e0f7eefafc02c0f5dce32b9303aa2a/jopt-simple-
 5.0.3.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.apache.logging.log4j/log4j-
 core/2.10.0/c90b597163cd28ab6d9687edd53db601b6ea75a1/log4j-core-
 2.10.0.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.apache.logging.log4j/log4j-
 api/2.10.0/fec5797a55b786184a537abd39c3fa1449d752d6/log4j-api-
 2.10.0.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/commons-beanutils/commons-
 beanutils/1.9.3/c845703de334ddc6b4b3cd26835458cb1cba1f3d/commons-beanutils-
 1.9.3.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/io.github.lukehutch/fast-
 classpath-scanner/2.0.11/ae34a7a5e6de8ad1f86e12f6f7ae1869fcfe9987/fast-classpath-
 scanner-2.0.11.jar:/Users/jblum/.gradle/caches/modules-2/files-
 2.1/org.apache.geode/geode-
 common/1.2.1/9db253081d33f424f6e3ce0cde4b306e23e3420b/geode-common-
 1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.geode/geode-
 json/1.2.1/bdb4c262e4ce6bb3b22e0f511cfb133a65fa0c04/geode-json-

```
1.2.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.lucene/lucene-
analyzers-common/6.4.1/c6f0f593503080204e9d33189cdc59320f55db37/lucene-analyzers-
common-6.4.1.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/org.apache.lucene/lucene-
queryparser/6.4.1/1fc5795a072770a2c47dce11a3c85a80f3437af6/lucene-queryparser-
6.4.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.lucene/lucene-
queries/6.4.1/6de41d984c16185a244b52c4d069b00f5b2b120f/lucene-queries-
6.4.1.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/org.apache.lucene/lucene-
core/6.4.1/2a18924b9e0ed86b318902cb475a0b9ca4d7be5b/lucene-core-
6.4.1.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/com.fasterxml.jackson.core/jackson-
core/2.9.6/4e393793c37c77e042ccc7be5a914ae39251b365/jackson-core-
2.9.6.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/javax.transaction/javax.transaction-
api/1.2/d81aff979d603edd90dcd8db2abc1f4ce6479e3e/javax.transaction-api-
1.2.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/commons-logging/commons-
logging/1.2/4bfc12adfe4842bf07b657f0369c4cb522955686/commons-logging-
1.2.jar:/Users/jblum/.gradle/caches/modules-2/files-2.1/commons-collections/commons-
collections/3.2.2/8ad72fe39fa8c91eaaf12aadb21e0c3661fe26d5/commons-collections-
3.2.2.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/org.springframework.shell/spring-
shell/1.2.0.RELEASE/d94047721f292bd5334b5654e8600cef4b845049/spring-shell-
1.2.0.RELEASE.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/com.google.guava/guava/17.0/9c6ef172e8de35fd8d4d8783e4821e57cdef7445/guava-
17.0.jar:/Users/jblum/.gradle/caches/modules-2/files-
2.1/jline/jline/2.12/ce9062c6a125e0f9ad766032573c041ae8ecc986/jline-2.12.jar
org.springframework.geode.docs.example.app.server.SpringBootApacheGeodeCacheServerAppl
ication
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

```
[info 2018/06/24 21:42:28.183 PDT <main> tid=0x1] Starting
SpringBootApacheGeodeCacheServerApplication on jblum-mbpro-2.local with PID 41795
(/Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build/classes/main
started by jblum in /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-
docs/build)
```

```
[info 2018/06/24 21:42:28.278 PDT <main> tid=0x1] Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@6fa51cd4:
```

startup date [Sun Jun 24 21:42:28 PDT 2018]; root of context hierarchy

[warn 2018/06/24 21:42:28.962 PDT <main> tid=0x1] @Bean method PdxConfiguration.pdxDiskStoreAwareBeanFactoryPostProcessor is non-static and returns an object assignable to Spring's BeanFactoryPostProcessor interface. This will result in a failure to process annotations such as @Autowired, @Resource and @PostConstruct within the method's declaring @Configuration class. Add the 'static' modifier to this method to avoid these container lifecycle issues; see @Bean javadoc for complete details.

[info 2018/06/24 21:42:30.036 PDT <main> tid=0x1]

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership.

The ASF licenses this file to You under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<https://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Build-Date: 2017-09-16 07:20:46 -0700
Build-Id: abaker 0
Build-Java-Version: 1.8.0_121
Build-Platform: Mac OS X 10.12.3 x86_64
Product-Name: Apache Geode
Product-Version: 1.2.1
Source-Date: 2017-09-08 11:57:38 -0700
Source-Repository: release/1.2.1
Source-Revision: 0b881b515eb1dcea974f0f5c1b40da03d42af9cf
Native version: native code unavailable
Running on: /10.0.0.121, 8 cpu(s), x86_64 Mac OS X 10.10.5
Communications version: 65
Process ID: 41795
User: jblum
Current dir: /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Home dir: /Users/jblum
Command Line Parameters:
-ea
-Dspring.profiles.active=
-javaagent:/Applications/IntelliJ IDEA 17
CE.app/Contents/lib/idea_rt.jar=62866:/Applications/IntelliJ IDEA 17

```

CE.app/Contents/bin
-Dfile.encoding=UTF-8
Class Path:

/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/charsets.jar
...
Library Path:
/Users/jblum/Library/Java/Extensions
/Library/Java/Extensions
/Network/Library/Java/Extensions
/System/Library/Java/Extensions
/usr/lib/java
.
System Properties:
  PID = 41795
  ...
[info 2018/06/24 21:42:30.045 PDT <main> tid=0x1] Startup Configuration:
  ### GemFire Properties defined with api ###
disable-auto-reconnect=true
jmx-manager=true
jmx-manager-port=1099
jmx-manager-start=true
jmx-manager-update-rate=2000
log-level=config
mcast-port=0
name=SpringBootApacheGeodeCacheServerApplication
start-locator=localhost[10334]
use-cluster-configuration=false
### GemFire Properties using default values ###
ack-severe-alert-threshold=0
...

[info 2018/06/24 21:42:30.090 PDT <main> tid=0x1] Starting peer location for
Distribution Locator on localhost/127.0.0.1

[info 2018/06/24 21:42:30.093 PDT <main> tid=0x1] Starting Distribution Locator on
localhost/127.0.0.1

[info 2018/06/24 21:42:30.094 PDT <main> tid=0x1] Locator was created at Sun Jun 24
21:42:30 PDT 2018

[info 2018/06/24 21:42:30.094 PDT <main> tid=0x1] Listening on port 10334 bound on
address localhost/127.0.0.1

...

[info 2018/06/24 21:42:30.685 PDT <main> tid=0x1] Initializing region
_monitoringRegion_10.0.0.121<v0>1024

[info 2018/06/24 21:42:30.688 PDT <main> tid=0x1] Initialization of region
_monitoringRegion_10.0.0.121<v0>1024 completed

```

...

```
[info 2018/06/24 21:42:31.570 PDT <main> tid=0x1] CacheServer Configuration:  
port=40404 max-connections=800 max-threads=0 notify-by-subscription=true socket-  
buffer-size=32768 maximum-time-between-pings=60000 maximum-message-count=230000  
message-time-to-live=180 eviction-policy=none capacity=1 overflow directory=.  
groups=[] loadProbe=ConnectionCountProbe loadPollInterval=5000 tcpNoDelay=true
```

```
[info 2018/06/24 21:42:31.588 PDT <main> tid=0x1] Started  
SpringBootApacheGeodeCacheServerApplication in 3.77 seconds (JVM running for 5.429)
```

You can now connect to this server using *Gfsh*:


```
$ echo $GEMFIRE
/Users/jblum/pivdev/apache-geode-1.2.1
jblum-mbpro-2:lab jblum$
jblum-mbpro-2:lab jblum$ gfsh

-----
 / ____/ ____/ ____/ /____/ 
 / / __/ /___ /_____ / 
 / /_/_/ / ___/ _____ / 
 /_____/ _/ _____/ _/    1.2.1
```

Monitor and Manage Apache Geode

```
gfsfsh>connect
Connecting to Locator at [host=localhost, port=10334] ..
Connecting to Manager at [host=10.0.0.121, port=1099] ..
Successfully connected to: [host=10.0.0.121, port=1099]
```

```
gfsh>list members
```

Name	Id
SpringBootApacheGeodeCacheServerApplication	10.0.0.121(SpringBootApacheGeodeCacheServerApplication:41795)<ec><v0>:1024

```
gfsh>describe member --name=SpringBootApacheGeodeCacheServerApplication
Name      : SpringBootApacheGeodeCacheServerApplication
Id        :
10.0.0.121(SpringBootApacheGeodeCacheServerApplication:41795)<ec><v0>:1024
Host      : 10.0.0.121
Regions   :
PID       : 41795
Groups    :
Used Heap : 184M
Max Heap  : 3641M
Working Dir : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Log file   : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Locators   : localhost[10334]
```

```
Cache Server Information
Server Bind           :
Server Port          : 40404
Running              : true
Client Connections    : 0
```

Now, let's start some additional servers to scale-out our cluster.

To do so, you simply need to vary the name of the members we will add to our cluster as peers. Apache Geode requires that the members in a cluster be named and the names of each member in the cluster be unique.

Additionally, since we are running multiple instances of our `SpringBootApacheGeodeCacheServerApplication` class, which also embeds a `CacheServer` instance enabling cache clients to connect, we need to be careful to vary our ports used by the embedded services.

Fortunately, we do not need to run another embedded *Locator* or *Manager* service (we only need 1 in this case), therefore, we can switch profiles from non-clustered to using the Spring *"clustered"* profile, which includes different configuration (the `ClusterConfiguration` class) to connect another server as a peer member in the cluster, which currently only has 1 member as shown in the `list members Gfsh` command output above.

To add another server, set the member name and the `CacheServer` port to a different number with the following run profile configuration:

Run profile configuration for server 2

```
-server -ea -Dspring.profiles.active=clustered -Dspring.data.gemfire.name=ServerTwo  
-Dspring.data.gemfire.cache.server.port=41414
```

Notice that we explicitly activated the *"clustered"* Spring profile, which enables the configuration provided in the nested `ClusteredConfiguration` class while disabling the `LonerConfiguration` class.

This `ClusteredConfiguration` class is also annotated with `@UseLocators`, which sets the Apache Geode `locators` property to *"localhost[10334]"*. By default, it assumes the Locator process/service is running on *"localhost"*, listening on the default Locator port of *"10334"*. You can of course adjust your Locators endpoint if your Locators are running elsewhere in your network by using the *"locators"* attribute of the `@UseLocators` annotation.



It is common in production environments to run multiple Locators as a separate process. Running multiple Locators provides redundancy in case a Locator process fails. If all Locator processes in your network fail, don't fret, your cluster will not go down. It simply means no other members will be able to join the cluster, allowing you to scale your cluster out, nor will any clients be able to connect. Simply just restart the Locators if this happens.

Additionally, we set the `spring.data.gemfire.name` property to *"ServerTwo"* adjusting the name of our member when it joins the cluster as a peer.

Finally, we set the `spring.data.gemfire.cache.server.port` to *"41414"* to vary the `CacheServer` port used by *"ServerTwo"*. The default `CacheServer` port is *"40404"*. If we had not set this property before starting *"ServerTwo"* we would have hit a `java.net.BindException`.



Both the `spring.data.gemfire.name` and `spring.data.gemfire.cache.server.port` properties are well-known properties used by SDG to dynamically configure Apache Geode using a Spring Boot `application.properties` file or Java System properties. You can find these properties in the Annotation Javadoc in SDG's Annotation-based Configuration model. For instance, the `spring.data.gemfire.cache.server.port` property is documented [here](#). Most of the SDG annotations include corresponding properties that can be defined in `application.properties` and is explained in more detail [here](#).

After starting our second server, "ServerTwo", we should see similar output at the command-line, and in *Gfsh*, when we `list members` and `describe member` again:

Gfsh output after starting server 2

```
gfsh>list members
      Name                                     | Id
-----|-----
SpringBootApacheGeodeCacheServerApplication |
10.0.0.121(SpringBootApacheGeodeCacheServerApplication:41795)<ec><v0>:1024
ServerTwo                                   | 10.0.0.121(ServerTwo:41933)<v1>:1025

gfsh>describe member --name=ServerTwo
Name      : ServerTwo
Id        : 10.0.0.121(ServerTwo:41933)<v1>:1025
Host      : 10.0.0.121
Regions   :
PID       : 41933
Groups    :
Used Heap : 165M
Max Heap  : 3641M
Working Dir : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Log file   : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build
Locators   : localhost[10334]

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

When `list members`, we see "ServerTwo" and when we `describe "ServerTwo"`, we see that its `CacheServer` port is appropriately set to "41414".

If we add 1 more server, "ServerThree" using the following run configuration:

Add server 3 to our cluster

```
-server -ea -Dspring.profiles.active=clustered -Dspring.data.gemfire.name=ServerThree  
-Dspring.data.gemfire.cache.server.port=42424
```

Again, we will see similar output at the command-line and in Gfsh:

Gfsh output after starting server 3

```
gfsh>list members  


| Name                                        | Id                                                                         |
|---------------------------------------------|----------------------------------------------------------------------------|
| SpringBootApacheGeodeCacheServerApplication | 10.0.0.121(SpringBootApacheGeodeCacheServerApplication:41795)<ec><v0>:1024 |
| ServerTwo                                   | 10.0.0.121(ServerTwo:41933)<v1>:1025                                       |
| ServerThree                                 | 10.0.0.121(ServerThree:41965)<v2>:1026                                     |

  
gfsh>describe member --name=ServerThree  
Name      : ServerThree  
Id         : 10.0.0.121(ServerThree:41965)<v2>:1026  
Host       : 10.0.0.121  
Regions    :  
PID        : 41965  
Groups     :  
Used Heap  : 180M  
Max Heap   : 3641M  
Working Dir : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build  
Log file    : /Users/jblum/pivdev/spring-boot-data-geode/spring-geode-docs/build  
Locators    : localhost[10334]  
  
Cache Server Information  
Server Bind      :  
Server Port     : 42424  
Running         : true  
Client Connections : 0
```

Congratulations! You just started a small Apache Geode cluster, with 3 members, using Spring Boot from inside your IDE.

It is pretty simple to build and run a Spring Boot, Apache Geode **ClientCache** application that connects to this cluster. Simply include and use Spring Boot for Apache Geode.

Testing

[Spring Test for Apache Geode](#) is a new, soon to be released and upcoming project to help developers write both *Unit* and *Integration Tests* when using Apache Geode in a Spring context.

In fact, the entire [test suite](#) in Spring Boot for Apache Geode is based on this project.

All Spring projects integrating with Apache Geode will use this new test framework for all their testing needs, making this new test framework for Apache Geode a proven and reliable solution for all your Apache Geode application testing needs when using Spring as well.

Later on, this reference guide will include and dedicate an entire chapter on testing.

Examples

The definitive source of truth on how to best use Spring Boot for Apache Geode is to refer to the [Samples](#).

Additionally, you may refer to the [Temperature Service](#), Spring Boot application implementing a Temperature Sensor and Monitoring, Internet of Things (IOT) example. The example uses SBDG to showcase Apache Geode CQ, Function Implementations/Executions and positions Apache Geode as a *caching provider* in Spring's Cache Abstraction. It is a working, sophisticated and complete example, and is highly recommended as a good starting point for real-world use cases.

You may also refer to the [boot-example](#) from the *Contact Application* Reference Implementation (RI) for Spring Data for Apache Geode (SDG) as yet another example.

References

1. Spring Framework [Reference Guide](#) | [Javadoc](#)
2. Spring Boot [Reference Guide](#) | [Javadoc](#)
3. Spring Data Commons [Reference Guide](#) | [Javadoc](#)
4. Spring Data for Apache Geode [Reference Guide](#) | [Javadoc](#)
5. Spring Session for Apache Geode [Reference Guide](#) | [Javadoc](#)
6. Spring Test for Apache Geode [README](#)
7. Apache Geode [User Guide](#) | [Javadoc](#)