Spring Boot for Apache Geode & VMware Tanzu GemFire Reference Guide

1.2.10.RELEASE

Table of Contents

Welcome to *Spring Boot for Apache Geode*.

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 applications in a Spring context.

Secondarily, Spring Boot for Apache Geode aims to provide developers with a consistent experience whether building and running Spring Boot, Apache Geode 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'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 builds on this very foundation cemented in Spring Data for Apache Geode (SDG [4]) since the Spring Data Kay Release Train.

[4]Spring Data for Apache Geode is commonly known as SDG.

1. Introduction

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

- *Look-Aside Caching*, Apache Geode as a caching provider in Spring's Cache Abstraction.
- *System of Record* (SOR), persisting application state reliably in Apache Geode using Spring Data Repositories.
- *Transactions*, managing application state consistently with Spring Transaction Management and SDG[5] support for both Local Cache and Global JTA Transactions.
- *Distributed Computations*, run with Apache Geode's Function Executions framework and conveniently implemented and executed with SDG[452] POJO-based, annotation support for Functions.
- *Continuous Queries*, expressing interests in a stream of events, where applications are able to react to and process changes to data in near real-time using Apache Geode Continuous Query (CQ). Handlers are defined as simple Message-Driven POJOs (MDP) using Spring's Message Listener Container, which has been extended by SDG[452] with its configurable CQ support.
- *Data Serialization* with Apache Geode PDX, including first-class configuration and support in SDG[452].
- *Security*, including Authentication & Authorization as well as Transport Layer Security (TLS) using Apache Geode's Secure Socket Layer (SSL). Once again, SDG[452] includes first-class support for configuring Auth and SSL.
- *HTTP Session state management*, by including Spring Session for Apache Geode on your application's classpath.

While Spring Data for Apache Geode offers a simple, convenient and declarative approach to configure all these powerful Apache Geode features, Spring Boot for Apache Geode & Pivotal Gemfire makes it even easier to do as we will explore throughout this Reference Documentation.

2. Getting Started

In order to be immediately productive and as effective as possible using Spring Boot for Apache Geode, 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 (SDG[2]) to truly simplify the development of Apache Geode 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[452] 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[452] 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 based applications in the easiest, quickest and most reliable way possible. We accomplish this by combining Spring Data for Apache Geode'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 (SBDG).

> **Tip**
>
> Refer to the corresponding Sample Guide and Code to see Spring Boot for Apache Geode in action!

[2] Spring Data for Apache Geode is commonly known as SDG.

3. Using Spring Boot for Apache Geode

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

Maven.

```
<dependencies>
    <dependency>
        <groupId>org.springframework.geode</groupId>
        <artifactId>spring-geode-starter</artifactId>
        <version>1.2.10.RELEASE</version>
    </dependency>
</dependencies
```

Gradle.

```
dependencies {
    compile 'org.springframework.geode:spring-geode-starter:1.2.10.RELEASE'
}
```

> **Tip**
>
> To use VMware Tanzu GemFire in place of Apache Geode, simply change the `artifactId` from `spring-geode-starter` to `spring-gemfire-starter`.

4. Building ClientCache Applications

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

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

For example, you can begin building a Spring Boot, Apache Geode or Pivotal GemFire, `ClientCache` application with either the `spring-geode-starter` or `spring-gemfire-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 or Pivotal GemFire server running on `localhost`, listening on the default `CacheServer` port, `40404`.

By default, an Apache Geode or Pivotal GemFire 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.

> **Tip**
>
> To develop with `LOCAL` Regions, you only need to define 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`, data management policy which will cause the data to be sent/received to and from 1 or more servers, respectively.

> **Tip**
>
> Compare and contrast the above configuration with Spring Data for Apache Geode/Pivotal GemFire's 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 GemFire/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 a (possibly) existing cluster (of data servers) as a peer. 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 I refer to here.

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

Spring Boot, Apache Geode/Pivotal GemFire CacheServer Application.

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

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

> **Tip**
>
> An Apache Geode/Pivotal GemFire "server" is not necessarily a “CacheServer” capable of serving cache clients. It is merely a peer member in the GemFire/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 GemFire/Geode services, an embedded *Locator*, which allows clients or even other peers to "locate" servers in a cluster, as well as an embedded *Manager*, which allows the GemFire/Geode application process to be managed and monitored using *Gfsh*, GemFire/Geode’s shell tool:

Spring Boot, Apache Geode/Pivotal GemFire CacheServer Application with *Locator* and *Manager* services enabled.

```
@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 or Pivotal GemFire `CacheServer` application. These additional servers started in *Gfsh* know about the Spring Boot, GemFire/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 GemFire/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:

Spring Boot, Apache Geode/Pivotal GemFire CacheServer Application with *Locator* and *Manager* services enabled.

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

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

> **Tip**
>
> Notice I configured the `SpringBootApacheGeodeCacheServerApplication` class, `@CacheServerApplication` annotation, `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 (or Pivotal GemFire) server and the *Gfsh Locator* and *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* it!

> **Tip**
>
> 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.

> **Tip**
>
> See the Appendix, Section 19.5, “Running an Apache Geode or Pivotal GemFire 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 or Pivotal GemFire Locator is a location-based service, or alternatively and more typically, a standalone process enabling clients to "locate" a cluster of Apache Geode/Pivotal GemFire 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 benefit and you are encouraged to read the documentation for more details.

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

Embedded Locator Service.

```
@EnableLocator
@CacheServerApplication
@SpringBootApplication
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 additional clients. If all the Locators in the cluster go down, then it is safe to simply restart them after a thorough diagnosis.

> **Note**
>
> 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.

```
@LocatorApplication
@SpringBootApplication
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 a Apache Geode or Pivotal GemFire process and overriding the default `ClientCache` instance provided by SBDG out-of-the-box.

> **Note**
>
> If your `@SpringBootApplication` class is annotated with `@LocatorApplication`, then it can only be a `Locator` and not a `ClientCache`, `CacheServer` or peer `Cache` application.

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.

SpringBootApacheGeodeLocatorApplication 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 and Pivotal GemFire 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 or Pivotal GemFire 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.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::  (v2.2.0.BUILD-SNAPSHOT)

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
2019-09-01 11:02:50,007  INFO he.geode.internal.logging.LoggingSession:  82 -
-------------------------------------------------------------------------------

  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:
```

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.

SpringBootApacheGeodeCacheServerApplication 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 { }

}
```

Simply enable the "clustered" Profile by using a IDE Run 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 started Locators and Servers.

```
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 simply must be careful to vary the ports is all and name your peer members appropriately. With Spring, and Spring Boot for Apache Geode or Pivotal GemFire (SBDG) in particular, it really is that easy!

5. Auto-configuration

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

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

> **Note**
>
> 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!

> **Tip**
>
> You should read the chapter in Spring Boot's Reference Guide on Auto-configuration.

> **Tip**
>
> You should review the chapter in Spring Data for Apache Geode and Pivotal GemFire's (SDG) Reference Guide on Annotation-based Configuration. For a quick reference, or an overview of Annotation-based Configuration, see here.

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 mat 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`).

> **Tip**
>
> 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-confiugration* 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 { ... }
```

> **Warning**
>
> 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 or Pivotal GemFire `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.

> **Warning**
>
> 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-confiugration* classes, see here.

`@ClientCacheApplication`

> **Note**
>
> The `ClientCacheAutoConfiguration` class corresponds to the `@ClientCacheApplication` annotation.

SBDG starts with the opinion that application developers will primarily be building Apache Geode or Pivotal GemFire client applications using Spring Boot.

Technically, this means building Spring Boot applications with either an Apache Geode or Pivotal GemFire `ClientCache` instance connected to a dedicated cluster of Apache Geode or Pivotal GemFire 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 { ... }
```

> **Tip**
>
> Refer to SDG's Referene Guide for more details on Apache Geode or Pivotal GemFire cache applications, and client/server applications in particular.

`@EnableGemfireCaching`

> **Note**
>
> The `CachingProviderAutoConfiguration` class corresponds to the `@EnableGemfireCaching` annotation.

If you simply used the core Spring Framework to configure either Apache Geode or Pivotal GemFire 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:

Configuring caching using Spring Data Geode.

```
@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) {
    ...
  }
}
```

> **Tip**
>
> Refer to the documentation for more details.

`@EnableContinuousQueries`

> **Note**
>
> 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 and Pivotal GemFire'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).

> **Tip**
>
> Refer to the documentation for more details.

`@EnableGemfireFunctionExecutions` & `@EnableGemfireFunctions`

> **Note**
>
> 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
interface CustomerService {

    @Autowired
    private MyCustomerapplicationFunctions customerFunctions;

    public 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!

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

> **Tip**
>
> Refer to the documentation for more details.

`@EnableGemfireRepositories`

> **Note**
>
> 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:

Using the application-specific Repository.

```
package example.app.sevice;

@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.

> **Tip**
>
> Refer to the documentation for more details.

`@EnableLogging`

> **Note**
>
> 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 and Pivotal GemFire 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.properites.

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.

`@EnablePdx`

> **Note**
>
> 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!

> **Tip**
>
> Refer to the documentation for more details.

`@EnableSecurity`

> **Note**
>
> 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 or Pivotal GemFire 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
```

> **Note**
>
> 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.

> **Tip**
>
> Refer to the documentation for more details.

`@EnableSsl`

> **Note**
>
> 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.

> **Tip**
>
> Refer to the documentation for more details.

`@EnableGemFireHttpSession`

> **Note**
>
> The `SpringSessionAutoConfiguration` class corresponds to the `@EnableSsl` annotation.

Configuring Apache Geode or Pivotal GemFire 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>1.2.10.RELEASE</version>
</dependency>
```

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

> **Tip**
>
> Refer to the documentation for more details.

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.

> **Tip**
>
> Refer to the documentation for more details.

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 or Pivotal GemFire in Spring [Boot] applications.

> **Note**
>
> SDG refers to Spring Data for Apache Geode & Pivotal GemFire. SSDG refers to Spring Session for Apache Geode & Pivotal GemFire and SBDG refers to *Spring Boot for Apache Geode & Pivotal GemFire*, this project.

> **Tip**
>
> 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`

> **Tip**
>
> 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 GemFire `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 GemFire/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 either Apache Geode or Pivotal GemFire in Spring Boot applications.

`@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.

When a cluster of Apache Geode or Pivotal GemFire 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 a 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.

> **Tip**
>
> Refer to the section in the SDG Reference Guide on Configuring Cluster Configuration Push for more details.

`@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.

```
@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:

Creating a client Region using Spring XML.

```
<gfe:client-region id="Customers" shorcut="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("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.

> **Tip**
>
> 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 sometMethod() {

    EntityType entity = ...;

    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:

Using a server-side Region on the client with a template.

```
@Component
class SomeApplicationComponent {

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

  public void sometMethod() {

    EntityType entity = ...;

    this.serverRegionTemplate.put(1, entity);

    ...
  }
```

> **Tip**
>
> Refer to the SDG Reference Guide on Configuring Cluster-defined Regions for more details.

`@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.

> **Note**
>
> 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.

> **Note**
>
> Keep in mind that OQL Indexes are not persistent between restarts (i.e. Apache Geode & Pivotal GemFire 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.

> **Tip**
>
> Refer to the SDG Reference Guide on Configuring Indexes for more details.

`@EnableExpiration` (SDG)

It is often useful to define both *Eviction* and *Expiration* policies, particularly with a system like Apache Geode or Pivotal GemFire, 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.

> **Tip**
>
> 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.

> **Tip**
>
> 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 the 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 { ... }
```

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

> **Tip**
>
> Refer to the SDG Reference Guide for more details on configuring Expiration, along with Annotation-based Data Expiration in particular.

`@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 and Pivotal GemFire 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 either Apache Geode or Pivotal GemFire. This is where the Spring Test for Apache Geode & Pivotal GemFire (STDG) project can help, particularly with *Unit Testing*.

For example, if you do not care what Apache Geode or Pivotal GemFire 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 or Pivotal GemFire's 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 or Pivotal GemFire 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 or Pivotal GemFire objects even works for GemFire/Geode 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 or Pivotal GemFire objects.

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

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

  @Test
  public void someTestCase() {

    Customer jonDoe = ...;

    // 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.

7. Externalized Configuration

Like Spring Boot itself (see here), Spring Boot for Apache Geode and Pivotal GemFire (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 & Pivotal GemFire provides a hierarchy of classes used to capture configuration for several Apache Geode or Pivotal GemFire 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 / Pivotal GemFire.

```
# Spring Boot application.properties used to configure Apache Geode

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 `Enviornment.

```
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 either Apache Geode or Pivotal GemFire 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 either Apache Geode or Pivotal GemFire 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 Apache Geode 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`
  }

  ...
}
```

8. Caching using Apache Geode or VMware GemFire

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

> **Tip**
>
> See the *Spring Data for Apache Geode Reference Guide* for more details on the support and configuration of Apache Geode or VMware GemFire as a *caching provider* in Spring's Cache Abstraction.

> **Tip**
>
> Make sure you thoroughly understand the concepts behind Spring's Cache Abstraction before you continue.

> **Tip**
>
> You can also refer to the relevant section on Caching in *Spring Boot's Reference Guide*. *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 and multi-site (WAN) capabilities, then you should consider Apache Geode, or alternatively VMware GemFire. Additionally, Pivotal Software, Inc. offers VMware GemFire as a service, known as Pivotal Cloud Cache (PCC), when deploying and running your Spring Boot applications in Pivotal Cloud Foundry (PCF).

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

> **Tip**
>
> 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 not likely 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 processEligility(Person person, Timespan timespan) {
        ...
    }
}
```

Notice the `@Cacheable` annotation 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 value to the caller.

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

If Spring Boot for Apache Geode/VMware GemFire detects that another *cache provider* has already been configured, then neither Apache Geode nor VMware GemFire will function as the *caching provider*. This allows users to configure, another store, e.g. Redis, as the *caching provider* and use Apache Geode or VMware GemFire as your application's persistent store, perhaps.

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 or VMware GemFire, this means a `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 or VMware GemFire's `@EnableCachingDefinedRegions` annotation.

The complete Spring Boot application looks like this:

```
package example.app;

import ...;

@SpringBootApplication
@EnableCachingDefinedRegions
class FinancialLoanApplication {

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

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

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

> **Note**
>
> Spring Boot for Apache Geode/VMware GemFire 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.

8.1 Look-Aside Caching, Near Caching, Inline Caching and Multi-Site Caching

Four different types of caching patterns can be applied with Spring when using Apace Geode or VMware GemFire 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 network hops).

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.

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.

Look-Aside Caching

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 findByAcccount(Account account) {

    // pre-processing logic here

    Customer customer = customerRepository.findByAccoundNumber(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.

> **Note**
>
> 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.

Near Caching

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 & VMware GemFire provides an *auto-configured*, `ClientCache` instance, out-of-the-box, by default. The `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 or VMware GemFire, 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:

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

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

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

> **Tip**
>
> Also see the Apache Geode documentation concerning Client/Server Event Distribution and specifically, "*Client Interest Registration on the Server*" when using local, 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.

Inline Caching

The final 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 this pattern of caching: Synchronous (*Read/Write-Through*) and Asynchronous (*Write-Behind*).

> **Note**
>
> Asynchronous (currently) only offers write capabilities, from the cache to the backend, external data source. There is not option to asynchronously and automatically load the cache when the entry value becomes available in the backend, external data source.

Synchronous Inline Caching

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

With Apache Geode and VMware GemFire, the cache, or using Apache Geode/VMware GemFire 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 store like Apache Cassandra, MongoDB or Neo4j).

> **Tip**
>
> See the Apache Geode User Guide on Data Loaders for more details.

Likewise, an Apache Geode or VMware GemFire Region can also be configured with a CacheWriter. A `CacheWriter` is responsible for writing any 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 some level of consistency between the backend data store and the Apache Geode or VMware GemFire Region.

> **Tip**
>
> 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 locic 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 or VMware GemFire 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:

DecisionManagementSystemLoader.

```
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.

     return ...
   }
}
```

> **Tip**
>
> 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.

DecisionManagementSystemWriter.

```
class DecisionManagementSystemWriter implements CacheWriter<?, EligibilityDecision> {

  private final DataSource dataSource;

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

  public void beforeCreate(EntryEvent<?, EligiblityDecision> entryEvent) {
    // Use configured DataSource to save (e.g. INSERT) the entry value into the backend data store
  }

  public void beforeUpdate(EntryEvent<?, EligiblityDecision> entryEvent) {
    // Use the configured DataSource to save (e.g. UPDATE or UPSERT) the entry value into the backend
 data store
  }

  public void beforeDestroy(EntryEvent<?, EligiblityDecision> entryEvent) {
    // Use the configured DataSource to delete (i.e. DELETE) the entry value from the backend data store
  }

  ...
}
```

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

> **Note**
>
> 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 & VMware GemFire (SBDG) now offers dedicated support and configuration of *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, that does not mean you have to give up Apache Geode or VMware GemFire 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, multi-site, low-latency/high-throughput scale-out application use cases.

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

And now, SBDG has just made this even easier.

EXAMPLE

Let's say you are using JPA/Hibernate to access (store and retrieve) 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.

> **Tip**
>
> 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.

> **Tip**
>
> 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.

8.2 Advanced Caching Configuration

Both Apache Geode and VMware GemFire support 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, both Apache Geode and VMware GemFire store data in the JVM Heap.

Several techniques can be employed to more effectively manage memory, such as using Eviction, possibly overflowing 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 & VMware GemFire make all of these configuration options available and simple to use.

8.3 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 either Apache Geode or VMware GemFire. 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.

> **Tip**
>
> 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 either Apache Geode or VMware GemFire as your application's persistent store (*System of Record*).

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

9. 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 comprise 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/Pivotal GemFire (SDG) offers the GemfireTemplate.

The `GemfireTemplate` provides a highly consistent and familiar API to perform data access operations on Apache Geode or Pivotal GemFire 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 & Pivotal GemFire (SBDG) will auto-configure `GemfireTemplate` beans for each Region present in the GemFire/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.

9.1 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.

9.2 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 GemFireConfiguration {
  ...
}
```

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.

9.3 Caching-defined Regions

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

When you are using Spring Framework's Cache Abstraction backed by either Apache Geode or Pivotal GemFire, 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.

> **Warning**
>
> 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.

9.4 Native-defined Regions

SBDG will even auto-configure `GemfireTemplate` beans for Regions defined using Apache Geode and Pivotal GemFire native configuration meta-data, such as `cache.xml`.

Given the following GemFire/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.

9.5 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 GemFireConfiguration {

  @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.

> **Note**
>
> 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.

10. Spring Data Repositories

Using Spring Data Repositories with Apache Geode or Pivotal GemFire makes short work of data access operations when using either Apache Geode or Pivotal GemFire 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 & Pivotal GemFire *auto-configures* the Spring Data for Apache Geode/Pivotal GemFire 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 GemFire/Geode "Customers" Region using the SDG `@Region` mapping annotation:

`Customer` entity class.

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

import ...;
@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 peristing and accessing `Customers`.

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

import ...;

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;

import ...;

@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/Pivotal GemFire Repositories extension in particular, for more details.

11. Function Implementations & Executions

11.1 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, both Apache Geode and Pivotal GemFire address this very important application concern in its Function Execution framework.

Spring Data for Apache Geode/Pivotal GemFire builds on this Function Execution framework by enabling developers to implement and execute GemFire/Geode Functions using a very simple POJO-based, annotation configuration model.

> **Tip**
>
> See here for the difference between Function implementation & executions.

Taking this 1 step further, Spring Boot for Apache Geode/Pivotal GemFire *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.

11.2 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 GemFire/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:

Function use.

```
@Service
class FinancialLoanApplicationService {

    private final EligibilityDeterminationExecution execution;

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

    @Cacheable("EligibilityDecisions", ...)
    EligibilityDecision processEligility(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/Pivotal GemFire handles the rest.

> **Tip**
>
> It is common to implement and register your application Functions on the server and execute them from the client.

12. 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 & Pivotal GemFire 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 & Pivotal GemFire 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 & Pivotal GemFire 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/Pivotal GemFire 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/Pivotal GemFire, 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.

13. 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/ Pivotal GemFire 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 or Pivotal GemFire.

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 or Pivotal GemFire 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 or Pivotal GemFire'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 and Pivotal GemFire maintain 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 GemFire/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/Pivotal GemFire.

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 GemFire/Geode's multi-site topology.

EligibilityDecision is automatically serialiable without implementing Java Serializable.

```
@Region("EligibilityDecisions")
class EligibilityDecision {
    ...
}
```

> **Tip**
>
> Apache Geode/Pivotal GemFire does support the standard Java Serialization format.

13.1 SDG `MappingPdxSerializer` vs. GemFire/Geode's `ReflectionBasedAutoSerializer`

Under-the-hood, Spring Boot for Apache Geode/Pivotal GemFire enables and uses Spring Data for Apache Geode/Pivotal GemFire's MappingPdxSerializer to serialize your application domain objects using PDX.

> **Tip**
>
> Refer to the SDG Reference Guide for more details on the `MappingPdxSerializer` class.

The `MappingPdxSerializer` offers several advantages above and beyond GemFire/Geode's own ReflectionBasedAutoSerializer class.

> **Tip**
>
> 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 GemFire/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/Pivotal GemFire 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;
import ...;

@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;
    }
}
```

Tip

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.

14. Security

This sections covers Security configuration for Apache Geode & Pivotal GemFire, which includes both Authentication & Authorization (collectively, Auth) as well as Transport Layer Security (TLS) using SSL.

> **Note**
>
> Securing Data at Rest is not generally supported by either Apache Geode, Pivotal GemFire or Pivotal Cloud Cache (PCC) yet.

14.1 Authentication & Authorization

Apache Geode & Pivotal GemFire 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 & Pivotal GemFire (SDG) provides first-class support for Apache Geode & Pivotal GemFire'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.

> **Note**
>
> Eventually, support and integration with Spring Security will be provided by SBDG as well.

When you use Spring Boot for Apache Geode & Pivotal GemFire (SBDG), which builds on the bits provided in Spring Data for Apache Geode & Pivotal GemFire (SDG), it makes short work of enabling Auth in both your clients and servers.

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 & Pivotal GemFire servers in the cluster will automatically be configured with Authentication and Authorization enabled.

Alternatively, you can provide an custom, application-specific implementation of Apache Geode & Pivotal GemFire's SecurityManager interface, declared and registered as a bean in the Spring `ApplicationContext`:

Declaring a custom Apache Geode or Pivotal GemFire `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 or Pivotal GemFire cluster with Authentication and Authorization enabled.

> **Tip**
>
> The Spring team recommends that you use Apache Shiro to manage the Authentication & Authorization of your Apache Geode or Pivotal GemFire servers over implementing Apache Geode or Pivotal GemFire's `SecurityManager` interface.

Auth for Clients

When Apache Geode or Pivotal GemFire servers have been configured with Authentication & Authorization enabled, then clients must authenticate when connecting.

Spring Boot for Apache Geode & Pivotal GemFire (SBDG) 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 managed environment, like Pivotal CloudFoundry (PCF).

Non-Managed Auth for Clients

To enable Auth for clients connecting to a secure Apache Geode or Pivotal GemFire 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 & Pivotal GemFire (SBDG) 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 & Pivotal GemFire (SBDG) 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.

14.2 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, both Apache Geode & Pivotal GemFire support 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 & Pivotal GemFire (SDG) 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 & Pivotal GemFire require 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 & Pivotal GemFire (SBDG) 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 & Pivotal GemFire (SBDG) will automatically configure your client to use SSL Socket connections.

If you are using Spring Boot to configure and bootstrap an Apache Geode or Pivotal GemFire server:

Spring Boot configured and bootstrapped Apache Geode or Pivotal GemFire server.

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

Then, Spring Boot will apply the same procedure to enable SSL on the servers, between peers, as well.

> **Tip**
>
> 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`.

14.3 Securing Data at Rest

Currently, neither Apache Geode nor Pivotal GemFire along with Spring Boot or Spring Data for Apache Geode and Pivotal GemFire 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 or Pivotal GemFire, 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.

15. Spring Boot Actuator

Spring Boot for Apache Geode and Pivotal GemFire (SBDG) adds Spring Boot Actuator support and dedicated `HealthIndicators` for Apache Geode and Pivotal GemFire. Equally, the provided `HealthIndicators` will even work with Pivotal Cloud Cache, which is backed by Pivotal 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 or Pivotal GemFire 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 / Pivotal GemFire 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/Pivotal GemFire, 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/Pivotal GemFire, out-of-the-box.

15.1 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 both Apache Geode and Pivotal GemFire, the cache instance is either a peer `Cache` instance, which makes your Spring Boot application part of a GemFire/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.

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 15.1. 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 15.2. 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 15.3. 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.
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.

Name	Description
geode.distributed-system.security-properties-location	Location of the security configuration properties.

Table 15.4. 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.

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 15.5. Region Details

Name	Description
geode.cache.regions.<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).
geode.cache.regions.<name>.data-policy	Policy used to manage the data in the Region (e.g. PARTITION, REPLICATE, etc).
geode.cache.regions.<name>.initial-capacity	Initial number of entries that can be held by a Region before it needs to be resized.
geode.cache.regions.<name>.load-factor	Load factor used to determine when to resize the Region when it nears capacity.

Name	Description
geode.cache.regions.<name>.key-constraint	Type-constraint for Region keys.
geode.cache.regions.<name>.off-heap	Determines whether this Region will store values in off-heap memory (NOTE: Keys are always kept on Heap).
geode.cache.regions.<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.regions.<name>.scope	Determines the `Scope` of the Region, which plays a factor in the Regions consistency-level, as it pertains to acknowledgements for writes.
geode.cache.regions.<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 15.6. Partition Region Details

Name	Description
geode.cache.regions.<name>.partition.colocated-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.regions.<name>.partition.local-max-memory	Total amount of Heap memory allowed to be used by this Region on this node.
geode.cache.regions.<name>.partition.redundant-copies	Number of replicas for this PARTITION Region, which is useful in High Availability (HA) use cases.
geode.cache.regions.<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.
geode.cache.regions.<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 15.7. Region Statistic Details

Name	Description
geode.cache.regions.<name>.statistics.hit-count	Number of hits for a Region entry.
geode.cache.regions.<name>.statistics.hit-ratio	Ratio of hits to the number of `Region.get(key)` calls.
geode.cache.regions.<name>.statistics.last-accessed-time	For an entry, determines the last time it was accessed with `Region.get(key)`.

Name	Description
geode.cache.regions.<name>.statistics.last-modified-time	For an entry, determines the time a Region's entry value was last modified.
geode.cache.regions.<name>.statistics.miss-count	Returns the number of times that a `Region.get` was performed and no value was found locally.

GeodeIndexesHealthIndicator

The `GeodeIndexesHealthIndicator` provides details about the configured Region `Indexes` used in OQL query data access operations.

The following details are covered:

Table 15.8. 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.
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 15.9. 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.

Name	Description
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.

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 15.10. 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.

Name	Description
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.

15.2 `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.

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 15.11. 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 15.12. 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 15.13. 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 GemFire/Geode Continuous Query system is tracked with the following, additional details on the client:

Table 15.14. 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).
geode.continuous-query.number-of-closed	Total number of closed CQs (if available).

Name	Description
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).

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 15.15. 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.
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.

Name	Description
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.
geode.pool.<name>.thread-local-connections	Thread local connection policy for this Pool.

15.3 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.

> **Note**
>
> The default cache instance created by Spring Boot for Apache Geode/Pivotal GemFire is a `ClientCache` instance.

> **Tip**
>
> 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.

GeodeCacheServersHealthIndicator

The `GeodeCacheServersHealthIndicator` provides details about the configured Apache Geode/ Pivotal GemFire `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 15.16. 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.
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>.maximum-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 `ServerSocket` 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.

Name	Description
geode.cache.server.<index>.tcp-no-delay	Configures the TCP/IP TCP_NO_DELAY 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 15.17. 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.connection-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.
geode.cache.server.<index>.load.subscription-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.

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 15.18. 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.

Name	Description
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.
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.

Name	Description
geode.async-event-queue.<id>.size	Number of entries in this queue.

GeodeGatewayReceiversHealthIndicator

The `GeodeGatewayReceiversHealthIndicator` provide details about the configured (WAN) `GatewayReceivers`, which are capable of receiving events from remote clusters when using Apache Geode/Pivotal GemFire's multi-site, WAN topology.

This `HealthIndicator` captures configuration meta-data along with the running state for each `GatewayReceiver`:

Table 15.19. 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 `ServerSocket` 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.

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/Pivotal GemFire's multi-site, WAN topology.

This `HealthIndicator` captures essential configuration meta-data and runtime characteristics for each `GatewaySender`:

Table 15.20. GatewaySender Details

Name	Description
geode.gateway-sender.count	Total number of configured GatewaySenders.
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.

Name	Description
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.
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.

16. Spring Session

This section covers auto-configuration of Spring Session using either Apache Geode or Pivotal GemFire 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 proving 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 & Pivotal GemFire provides auto-configuration support to configure either Apache Geode or Pivotal GemFire as the user's session information management provider and store when Spring Session for Apache Geode or Pivotal GemFire is on your Spring Boot application's classpath.

> **Tip**
>
> You can learn more about Spring Session for Apache Geode & Pivotal GemFire in the docs.

16.1 Configuration

There is nothing special that you need to do in order to use either Apache Geode or Pivotal GemFire 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>2.2.5.RELEASE</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>1.2.10.RELEASE</version>
</dependency>
```

> **Tip**
>
> You may replace Apache Geode with Pivotal Cloud Cache or Pivotal GemFire by changing the artifact ID from `org.springframework.session:spring-session-data-geode`

> to `org.springframework.session:spring-session-data-gemfire`. Alternatively, you may replace Apache Geode with Pivotal Cloud Cache (PCC) or Pivotal GemFire by changing the artifact ID from `spring-geode-starter-session` to `spring-gemfire-starter-session`. The version number is the same.

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 either Apache Geode or Pivotal GemFire, or even Pivotal Cloud Cache.

16.2 Custom Configuration

By default, Spring Boot for Apache Geode & Pivotal GemFire (SBDG) applies reasonable and sensible defaults when configuring Apache Geode or Pivotal GemFire 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/Pivotal GemFire 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?

Custom Configuration using Properties

Spring Session for Apache Geode/Pivotal GemFire publishes well-known configuration properties for each of the various Spring Session configuration options when using Apache Geode or Pivotal GemFire 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 or Pivotal GemFire.

In addition to the properties provided in and by Spring Session for Apache Geode/Pivotal GemFire, Spring Boot for Apache Geode/Pivotal GemFire also recognizes and respects the `spring.session.timeout` property as well as the `server.servlet.session.timeout` property as discussed here.

> **Tip**
>
> `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.

Custom Configuration using a Configurer

Spring Session for Apache Geode/Pivotal GemFire 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 or Pivotal GemFire.

The `SpringSessionGemFireConfigurer`, when declared in the Spring `ApplicationContext`, takes precedence over any of the Spring Session (for Apache Geode/Pivotal GemFire) configuration properties, and will effectively override them when both are present.

More information on using the `SpringSessionGemFireConfigurer` can be found in the docs.

16.3 Disabling Session State Caching

There may be cases where you do not want your Spring Boot application to manage (HTTP) Session state using either Apache Geode or Pivotal GemFire. 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:

Use Redis as the Spring Session Provider.

```
#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.

> **Tip**
>
> 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 either Apache Geode or Pivotal GemFire as your application's persistent store (*System of Record*).

> **Note**
>
> Spring Boot does not properly recognize `spring.session.store-type=[gemfire|geode]` even though Spring Boot for Apache Geode/Pivotal GemFire is setup to handle either of these property values (i.e. either "gemfire" or "geode").

16.4 Using Spring Session with Pivotal Cloud Cache

Whether you are using Spring Session in a Spring Boot `ClientCache` application connecting to an externally managed cluster of Apache Geode or Pivotal GemFire servers, or connecting to a cluster of servers in a Pivotal Cloud Cache instance managed by a Pivotal Platform environment, the setup is the same.

Spring Session for Apache Geode, Pivotal GemFire, and Pivotal Cloud Cache (PCC) expects there to exist a cache Region in the cluster that will store and manage the (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*".

> **Note**
>
> 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 GemFire and 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.

> **Tip**
>
> 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 or Pivotal GemFire clusters, or when using PCC running in a Pivotal Platform environment.

17. Pivotal CloudFoundry

In most cases, when you "*push*" (i.e. "*deploy*") 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 is a managed version of Pivotal GemFire running in Pivotal CloudFoundry. 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 or Pivotal GemFir clusters. It handles many of the infrastructure-related, operational concerns so you do not have to.

17.1 Running Spring Boot applications as a specific user

By default, Spring Boot applications run as a "*cluster_operator*" Role-based user in Pivotal CloudFoundry (PCF) when the app is bound to a Pivotal Cloud Cache (PCC) 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 and Pivotal GemFire 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 application will be bound.

Configuration metadata for the PCC service instance might appear as follows:

Pivotal Cloud Cache configuration metadata.

```
{
  "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
```

> **Tip**
>
> 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 and Pivotal GemFire (SDG) to configure the runtime user of your Spring Data application when connecting to either an externally managed Apache Geode or Pivotal GemFire 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.

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.

17.2 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 & Pivotal GemFire (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.

17.3 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:

Multple 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]
```

> **Note**
>
> 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.

> **Tip**
>
> 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.

```
@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.

> **Note**
>
> 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.

17.4 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 or Pivotal GemFire cluster.

Spring Boot for Apache Geode & Pivotal GemFire (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.

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 (PCC) 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 or Pivotal GemFire 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.

```
Downloading Network Helper...
Progress: |====================>| 100.0%
Installing cfdevd network helper (requires administrator privileges)...
Password:
Setting up IP aliases for the BOSH Director & CF Router (requires administrator privileges)
Downloading Resources...
Progress: |====================>| 100.0%
Setting State...
WARNING: PCF Dev requires 8192 MB of RAM to run. This machine may not have enough free RAM.
Creating the VM...
Starting VPNKit...
Waiting for the VM...
Deploying the BOSH Director...

Deploying PAS...
  Done (14m34s)
Deploying Apps-Manager...
  Done (1m41s)

   #######  ###################### ###########   ###
   ##############################################   ###
   ###########     ######  ###  #########  ###   ###
   ####### ###     ######  ###  #########  #### ####
   ###     ###########     ################ #######
   ###      ##########     ####### ########  #####
               is now running!

  To begin using PCF Dev, please run:
      cf login -a https://api.dev.cfdev.sh --skip-ssl-validation

  Admin user => Email: admin / Password: admin
  Regular user => Email: user / Password: pass

  To access Apps Manager, navigate here: https://apps.dev.cfdev.sh

  To deploy a particular service, please run:
      cf dev deploy-service <service-name> [Available services: mysql,redis,rabbitmq,scs]
```

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 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.

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:

No Application Regions.

```
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.

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 (or Pivotal GemFire) 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>'
```

> **Note**
>
> 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 Crendentials 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`,

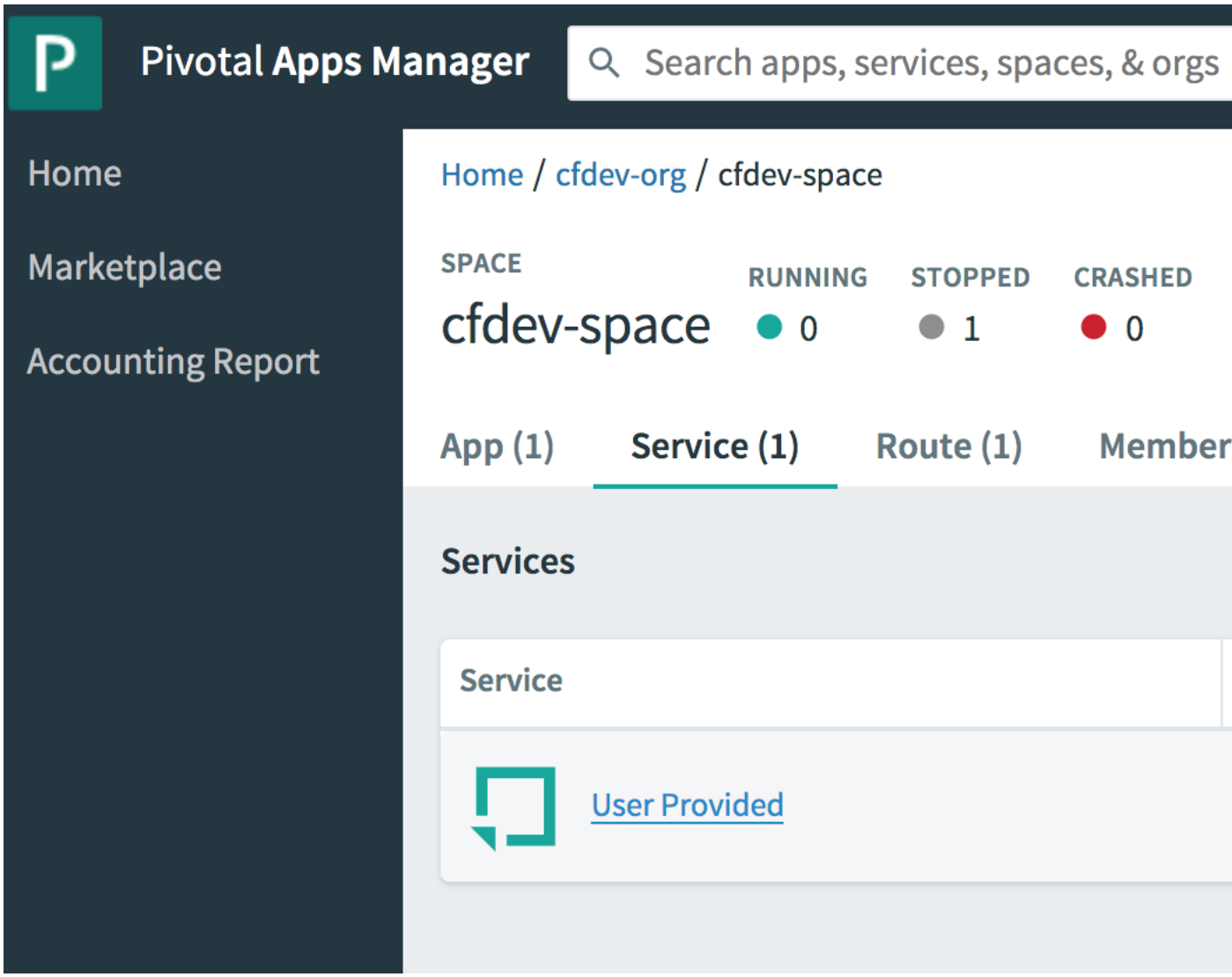
Finally, we set the `username` and `password` accordingly.

> **Tip**
>
> Spring Boot for Apache Geode (SBDG) provides template files in the /opt/jenkins/data/workspace/spring-boot-data-geode_1.2.x/spring-geode-docs/src/main/resources directory.

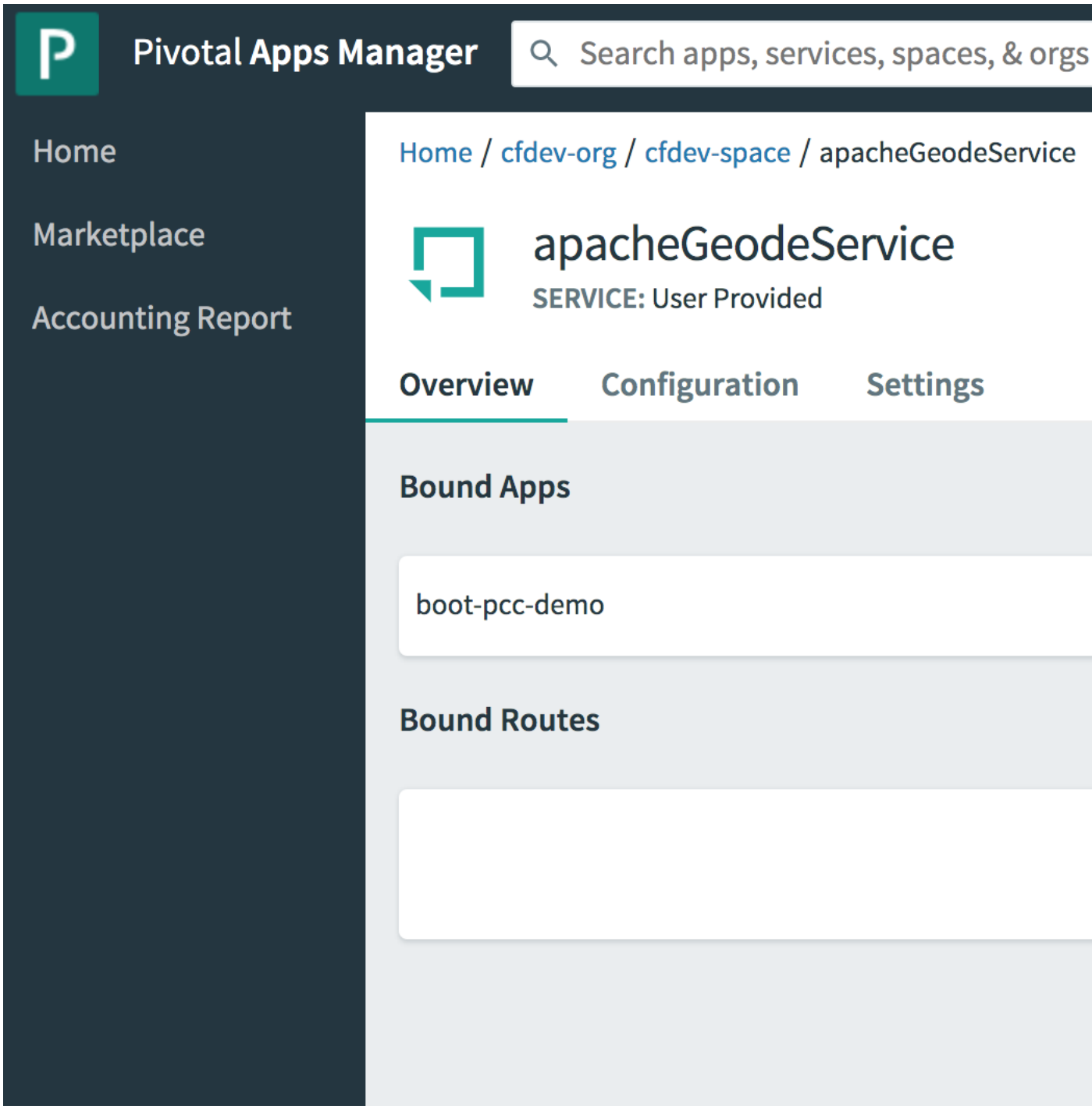
Once the service has been created, you can query the details from the `cf` CLI:

```
$ 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
```

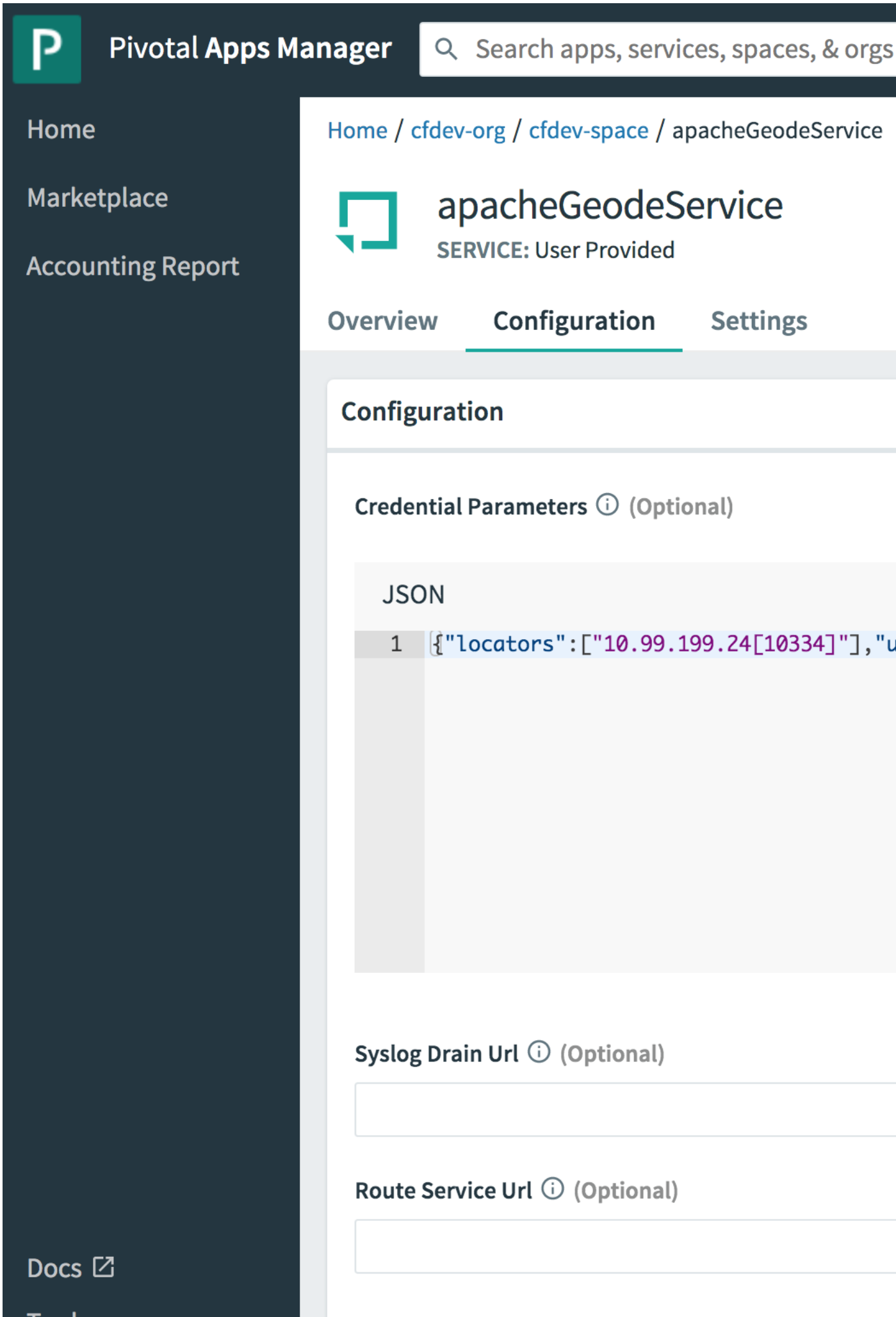
You can also view the "apacheGeodeService" from Apps Manager, starting from the `Service` tab in your org and space:



By clicking on the "apacheGeodeService" service entry in the table you can get all the service details, such the bound apps:



Configuration:


Pivotal Apps Manager
Search apps, services, spaces, & orgs
Home
Marketplace
Accounting Report
Home / cfdev-org / cfdev-space / apacheGeodeService
apacheGeodeService
SERVICE: User Provided
Overview
Configuration
Settings
Configuration
Credential Parameters (Optional)
JSON
1 {"locators":["10.99.199.24[10334]"],"u
Syslog Drain Url (Optional)
Route Service Url (Optional)
Docs

And so on.

> **Tip**
>
> You can learn more about CUPS in the PCF documentation, here.

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.

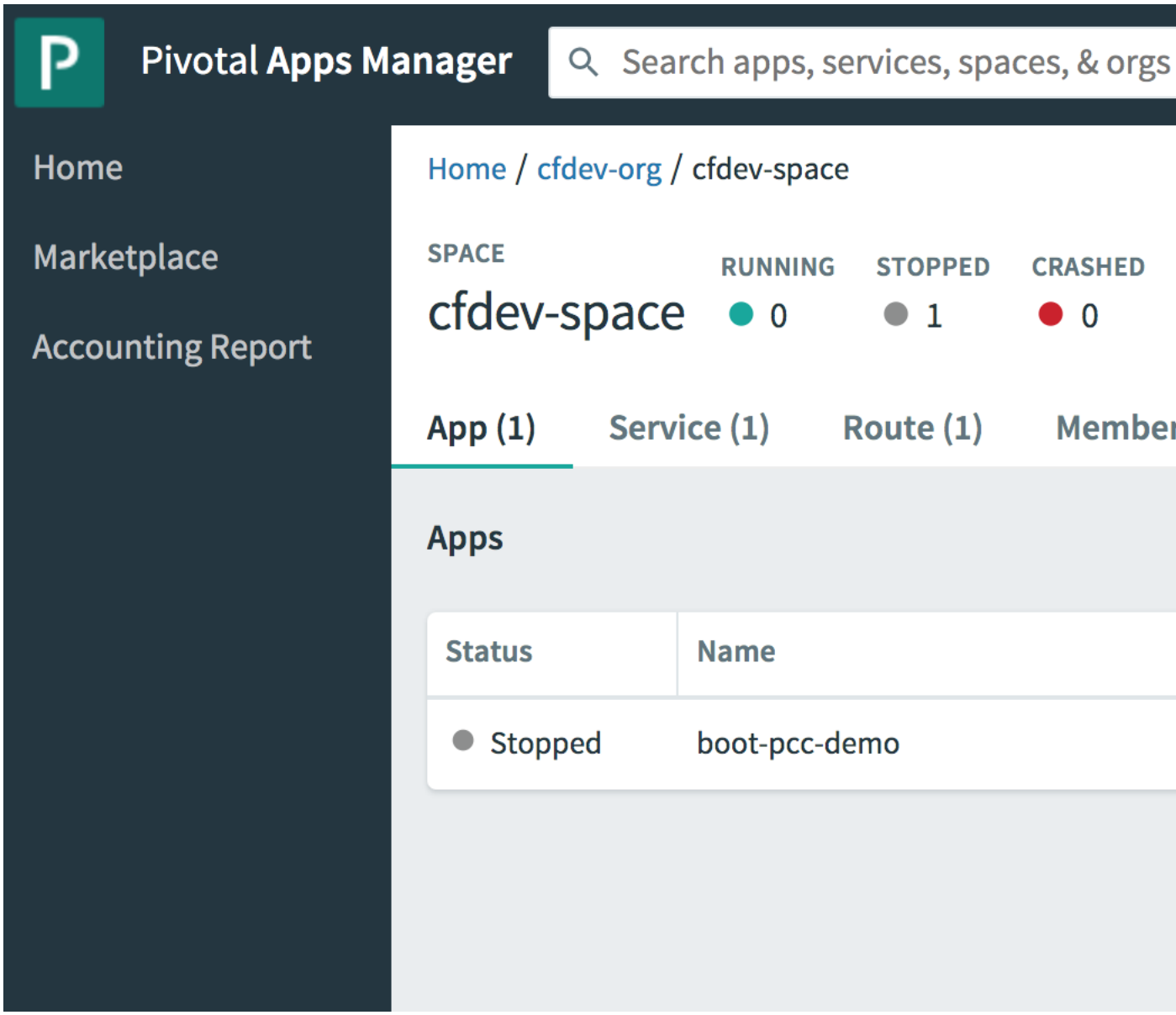
```
cf bind-service boot-pcc-demo apacheGeodeService
```

Or, alternatively, you can create a YAML file (`manifest.yml` in `src/main/resources`) containing the deployment descriptor:

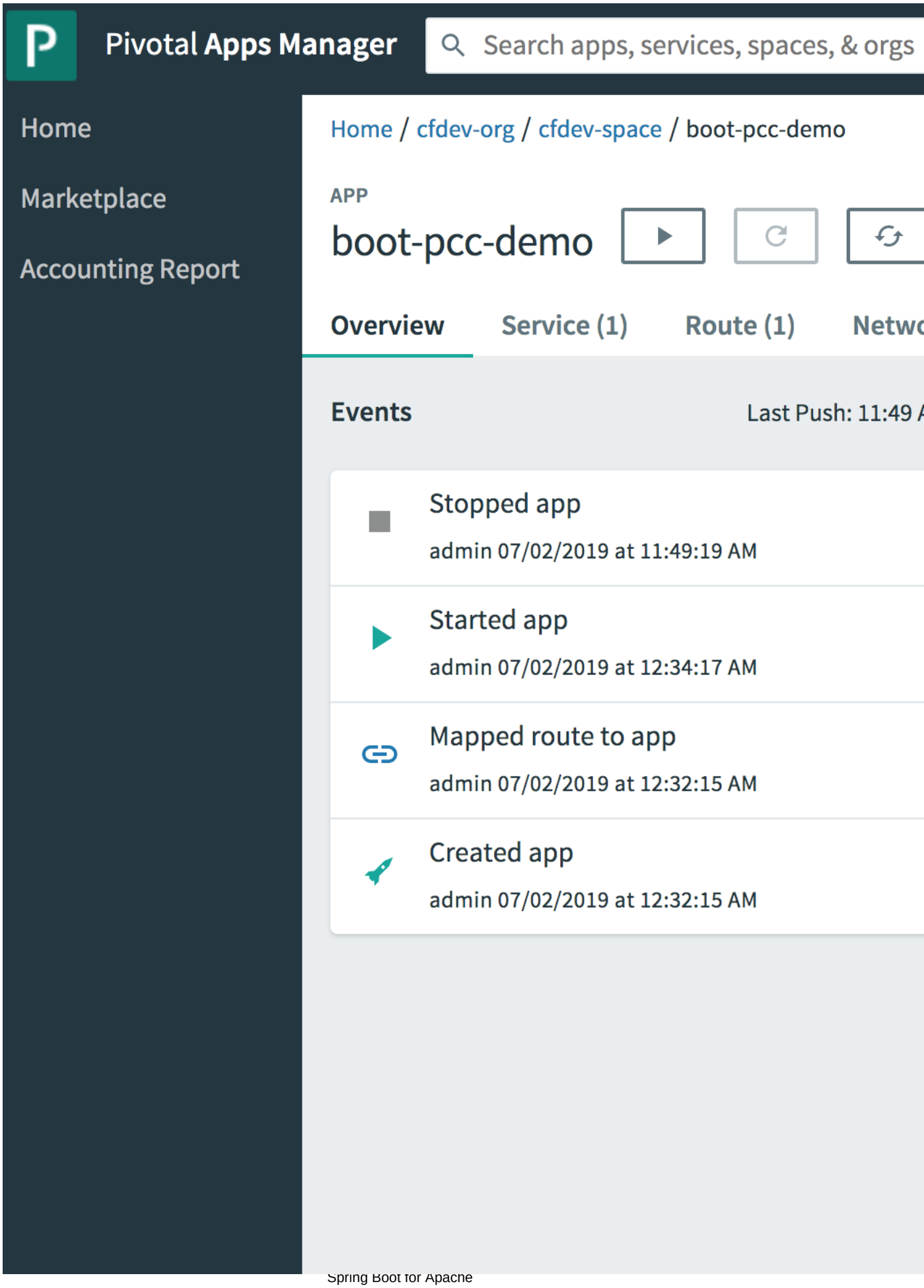
Example YAML deployment descriptor file.

```
\---
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
```

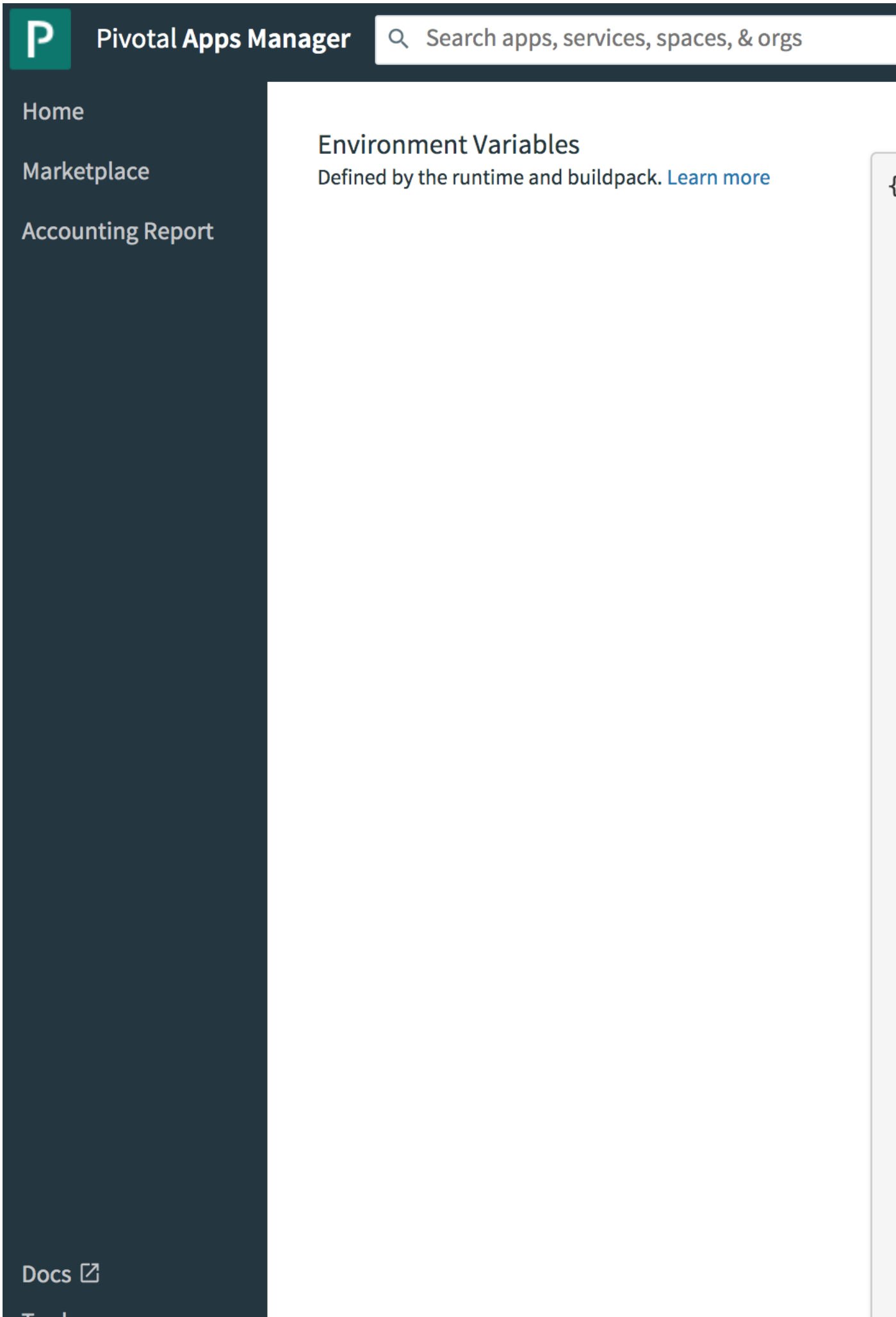
You can also use Apps Manager to view app details and un/bind additional services. Start by navigating to the App tab under your org and space:



From there, you can click on the desired app and navigate to the `Overview`:


Pivotal Apps Manager
Search apps, services, spaces, & orgs
Home
Marketplace
Accounting Report
Home / cfdev-org / cfdev-space / boot-pcc-demo
APP
boot-pcc-demo
Overview
Service (1)
Route (1)
Netwo
Events
Last Push: 11:49 A
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

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*:


Pivotal Apps Manager
Search apps, services, spaces, & orgs
Home
Marketplace
Accounting Report
Environment Variables
Defined by the runtime and buildpack. Learn more
Docs

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.

Running the Spring Boot application

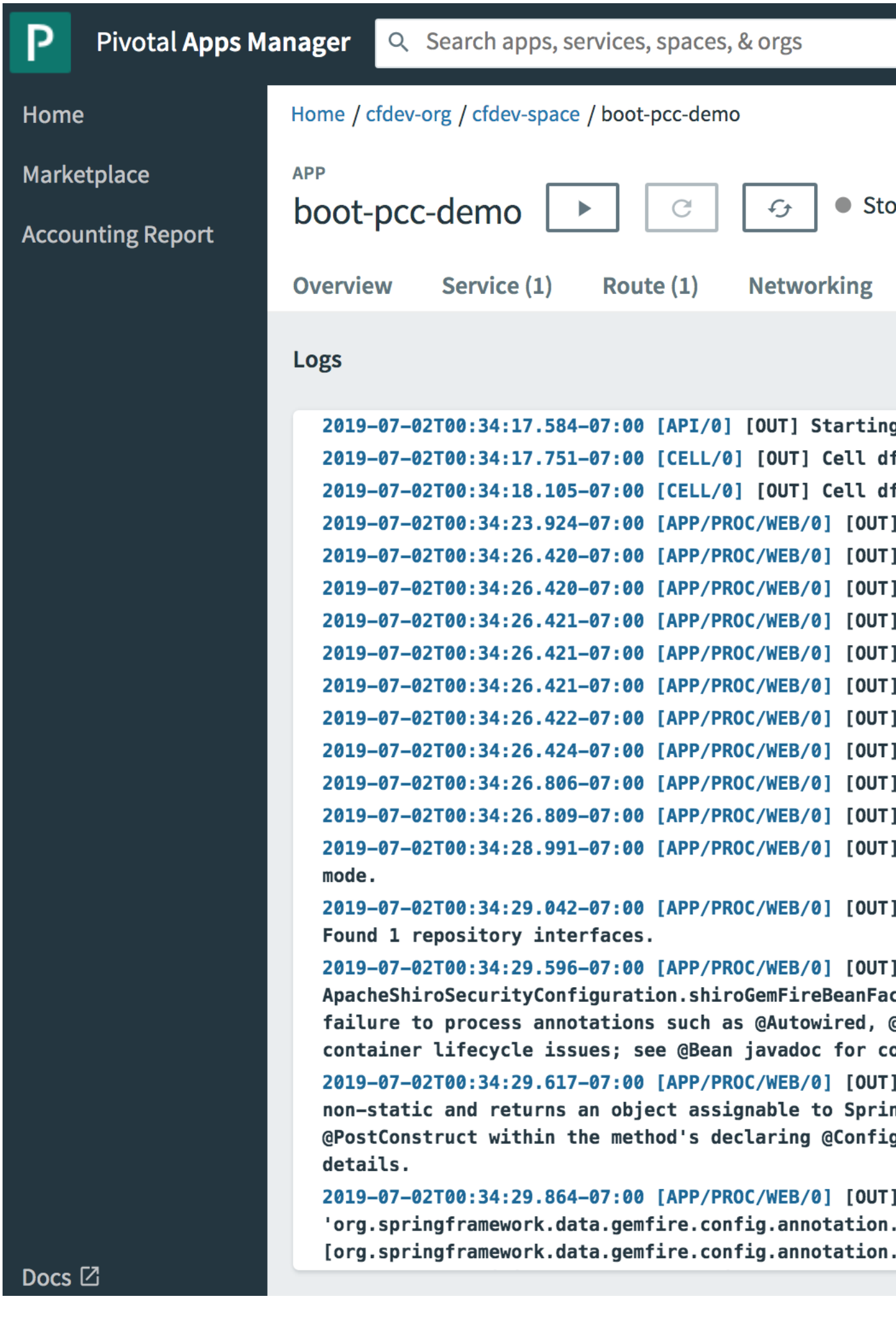
All that is left to do now is run the app.

You can start the PCCDemo app from the `cf` CLI using the following command:

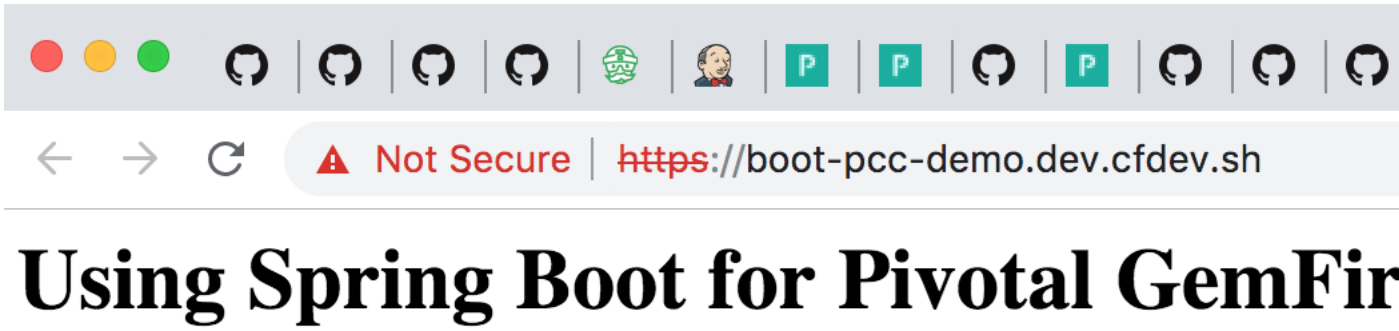
Start the Spring Boot app.

```
$ cf start boot-pcc-demo
```

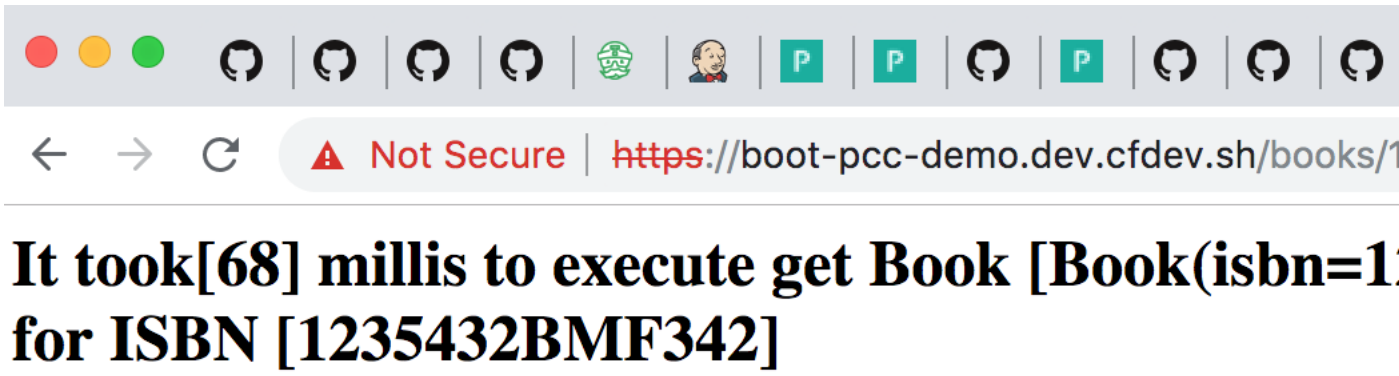
Alternatively, you can also start the app from Apps Manager. This is convenient since then you can tail and monitor the application log file.


Pivotal Apps Manager
Search apps, services, spaces, & orgs
Home
Marketplace
Accounting Report
Home / cfdev-org / cfdev-space / boot-pcc-demo
APP
boot-pcc-demo
Sto
Overview
Service (1)
Route (1)
Networking
Logs
2019-07-02T00:34:17.584-07:00 [API/0] [OUT] Starting
2019-07-02T00:34:17.751-07:00 [CELL/0] [OUT] Cell df
2019-07-02T00:34:18.105-07:00 [CELL/0] [OUT] Cell df
2019-07-02T00:34:23.924-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.420-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.420-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.421-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.421-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.421-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.422-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.424-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.806-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:26.809-07:00 [APP/PROC/WEB/0] [OUT]
2019-07-02T00:34:28.991-07:00 [APP/PROC/WEB/0] [OUT]
mode.
2019-07-02T00:34:29.042-07:00 [APP/PROC/WEB/0] [OUT]
Found 1 repository interfaces.
2019-07-02T00:34:29.596-07:00 [APP/PROC/WEB/0] [OUT]
ApacheShiroSecurityConfiguration.shiroGemFireBeanFac
failure to process annotations such as @Autowired, @
container lifecycle issues; see @Bean javadoc for co
2019-07-02T00:34:29.617-07:00 [APP/PROC/WEB/0] [OUT]
non-static and returns an object assignable to Sprin
@PostConstruct within the method's declaring @Config
details.
2019-07-02T00:34:29.864-07:00 [APP/PROC/WEB/0] [OUT]
'org.springframework.data.gemfire.config.annotation.
[org.springframework.data.gemfire.config.annotation.
Docs

Once the app has started, you can click the VIEW APP link in the upper right corner of the APP screen.



You can navigate to any of the application Web Service, Controller endpoints. For example, if you know the ISBN of a Book, you can access it from the Web browser:



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
```

17.5 Summary

There you have it!

The ability to deploy Spring Boot, Apache Geode or Pivotal GemFire `ClientCache` applications to Pivotal CloudFoundry, yet connect your app to a externally managed, standalone Apache Geode or Pivotal GemFire cluster.

Indeed, this is will be a useful arrangement and stepping stone for many users as they begin their journey towards a Cloud-Native platform like Pivotal CloudFoundry (PCF) and using services like Pivotal Cloud Cache (PCC).

Later, when the time comes and your need is very 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.

18. Samples

This section contains working examples demonstrating how to use Spring Boot for Apache Geode and Pivotal GemFire (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 / Pivotal GemFire, Spring Boot applications.

Table 18.1. 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/Pivotal GemFire	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/Pivotal GemFire	Explains how to use Spring Boot Actuator for Apache Geode and how it works.	Boot Actuator
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 **Look-Aside Caching** sample above.	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 **Look-Aside Caching** sample above and is the 3rd and final leg in our study on *caching patterns*.	Near Caching

Guide	Description	Source
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

19. Appendix

The following appendices provide additional help while developing Spring Boot applications backed by Apache Geode or Pivotal GemFire.

Table of Contents

19.1 Auto-configuration vs. Annotation-based configuration

The question most often asked is, "*What Spring Data for Apache Geode/Pivotal GemFire annotations can I use, or must I use, when developing Apache Geode or Pivotal GemFire 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 & Pivotal GemFire, which showcases the *auto-configuration* provided by Spring Boot for Apache Geode/ Pivotal GemFire in action.

Background

To help answer this question, we must start by reviewing the complete collection of available Spring Data for Apache Geode/Pivotal GemFire (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/Pivotal GemFire (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 GemFire/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 either Apache Geode or Pivotal GemFire. 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 GemFire/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 theses 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 or Pivotal GemFire as "client" applications in GemFire/Geode's client/server topology. This is especially true as users migrate their applications to a managed environment, such as Pivotal CloudFoundry (PCF) using Pivotal Cloud Cache (PCC).

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-Manage Envionment.

```
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 Pivotal CloudFoundry (PCF) when using Pivotal Cloud Cache (PCC), 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 PCF and bound to services (e.g. PCC).

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 GemFire/Geode, which in certain cases, like Security, not even Spring is able to completely shield users from the nuances of GemFire/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 meta-data must be supplied to GemFire/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 GemFire/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, GemFire/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 GemFireFunctions {

  @GemfireFunction
  Object exampleFunction(Object arg) {
    ...
  }
}
```

And then define the execution:

Function Execution.

```
@OnRegion(region = "Example")
interface GemFireFunctionExecutions {

  Object exampleFunction(Object arg);
}
```

SBDG will automatically find, configure and register Function Implementations (POJOs) in GemFire/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 GemFire/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 GemFire/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.

19.2 Configuration Metadata Reference

The following 2 reference sections cover documented and well-known properties recognized and processed by *Spring Data for Apache Geode/Pivotal GemFire* (SDG) as well as *Spring Session for Apache Geode/Pivotal GemFire* (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 or Pivotal GemFire 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 19.1. `spring.data.gemfire.*` properties

Name	Description	Default	From
name	Name of the Apache Geode / Pivotal GemFire member.	SpringBasedCacheClientApplication	CliApplication.name PeerCacheApplication.name
locators	Comma-delimited list of Locator	[]	PeerCacheApplication.locators

Name	Description	Default	From
	endpoints formatted as: locator1[port1],…,locatorN[portN].		
use-bean-factory-locator	Enable the SDG BeanFactoryLocator when mixing Spring config with GemFire/Geode native config (e.g. cache.xml) and you wish to configure GemFire objects declared in cache.xml with Spring.	false	ClientCacheApplication.useBeanFactoryLocator

Table 19.2. `spring.data.gemfire.*` GemFireCache properties

Name	Description	Default	From
cache.copy-on-read	Configure whether a copy of an object returned from Region.get(key) 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.criticalOffHeapPercentag
cache.enable-auto-region-lookup	Configure whether to lookup Regions configured in GemFire/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		ClientCacheApplication.evictionHeapPercentage

Name	Description	Default	From
	begin on Regions configured for HeapLRU eviction.		
cache.eviction-off-heap-percentage	Percentage of off-heap at or above which the eviction should begin on Regions configured for HeapLRU eviction.		ClientCacheApplication.evictionOffHeapPercenta
cache.log-level	Configure the log-level of an Apache Geode / Pivotal GemFire 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 GemFire/ 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 19.3. `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		ClientCacheApplication.durableClientId

Name	Description	Default	From
	used by servers to reestablish any messaging that was interrupted by client downtime.		
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 19.4. `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
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	60	PeerCacheApplication.lockTimeout

Name	Description	Default	From
	will wait to obtain a distributed lock lease.		
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 GemFire cache member node would pull it's configuration meta-data from the cluster-based Cluster Configuration Service.	false	PeerCacheApplication.useClusterConfiguration

Table 19.5. `spring.data.gemfire.*` CacheServer properties

Name	Description	Default	From
cache.server.auto-startup	Configures whether the CacheServer should be started automatically at runtime.	true	CacheServerApplication.autoStartup
cache.server.bind-address	Configures the IP address or hostname that this cache server will listen on.		CacheServerApplication.bindAddress

Name	Description	Default	From
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
cache.server.socket-buffer-size	Configures buffer size of the socket connection to this CacheServer.	32768	CacheServerApplication.socketBufferSize

Name	Description	Default	From
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.subscriptionDiskStoreN
cache.server.subscription-eviction-policy	Configures the eviction policy that is executed when capacity of the client subscription queue is reached.	none	CacheServerApplication.subscriptionEvictionPoli
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 19.6. `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.serverRegionShortcu

Table 19.7. `spring.data.gemfire.*` DiskStore properties

Name	Description	Default	From
disk.store.allow-force-compaction	Configures whether to allow DiskStore.forceCompaction() to be called on Regions using a DiskStore.	false	EnableDiskStore.allowForceCompaction
disk.store.auto-compact	Configures whether to cause the disk files to be	true	EnableDiskStore.autoCompact

Name	Description	Default	From
	automatically compacted.		
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 GemFire/ Geode DiskStore (oplog) files will be stored.	[]	EnableDiskStore.diskDirectories.location
disk.store.directory.size	Configures the amount of disk space allowed to store DiskStore (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
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	1000	EnableDiskStore.timeInterval

Name	Description	Default	From
	asynchronously is flushed to disk.		
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 19.8. `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 19.9. `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 19.10. `spring.data.gemfire.*` Logging properties

Name	Description	Default	From
logging.level	Configures the log-level of an Apache Geode / Pivotal GemFire 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 19.11. `spring.data.gemfire.*` Management properties

Name	Description	Default	From
management.use-http	Configures whether to use the HTTP protocol to communicate with a GemFire/Geode Manager.	false	EnableClusterConfiguration.useHttp
management.http.host	Configures the IP address or hostname of the GemFire/Geode Manager running the HTTP service.		EnableClusterConfiguration.host
management.http.port	Configures the port used by the GemFire/Geode Manager's HTTP service to listen for connections.	7070	EnableClusterConfiguration.port

Table 19.12. `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
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 th 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	2000	EnableManager.updateRate

Name	Description	Default	From
	will push updates to any JMX Managers.		

Table 19.13. `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
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 19.14. `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	5000	EnablePool.idleTimeout

Name	Description	Default	From
	(and closing) the connection.		
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
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	true	EnablePool.prSingleHopEnabled

Name	Description	Default	From
	hosting Regions with DataPolicy.PARTITION.		
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
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	100	EnablePool.subscriptionAckInterval

Name	Description	Default	From
	to the CacheServer for events received from the server subscriptions.		
pool.subscription-enabled	Configures whether the created Pool will have server-to-client subscriptions enabled.	false	EnablePool.subscriptionEnabled
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.subscriptionMessageTrackingTimeo
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 19.15. `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.clientAuthentiationInitializer
security.client.authenticator	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
security.log.file	Configures the pathname to a log file used for security log messages.		EnableAuth.securityLogFile

Name	Description	Default	From
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
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.securityPostProcessorClassNam
security.shiro.ini-resource-path	Configures the Apache Geode System Property referring to the location of an Apache Shiro INI file that configures		EnableSecurity.shiroIniResourcePath

Name	Description	Default	From
	the Apache Shiro Security Framework in order to secure Apache Geode.		

Table 19.16. `spring.data.gemfire.*` SSL properties

Name	Description	Default	From
security.ssl.certificate.aliases.cluster	Configures the alias to the stored SSL certificate used by the cluster to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.alias.default	Configures the default alias to the stored SSL certificate used to secure communications across the entire GemFire/Geode system.		EnableSsl.defaultCertificateAlias
security.ssl.certificate.aliases.gateway	Configures the alias to the stored SSL certificate used by the WAN Gateway Senders/ Receivers to secure communications.		EnableSsl.componentCertificateAliases
security.ssl.certificate.aliases.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.aliases.locator	Configures the alias to the stored SSL certificate used by the Locator to secure communications.		EnableSsl.componentCertificateAliases

Name	Description	Default	From
security.ssl.certificate.alias.servers	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 GemFire/ 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
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

Name	Description	Default	From
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 19.17. `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	false	EnableHttpService.startDeveloperRestApi

Name	Description	Default	From
	API web service. A full installation of Apache Geode or Pivotal GemFire is required and you must set the $GEODE environment variable.		
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
service.redis.bind-address	Configures the IP address or hostname of the system NIC used by the embedded Redis server to bind an 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 19.18. `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

Name	Description	Default	From
cache.client.region.shortcut	Configures the DataPolicy used by the client Region to manage (HTTP) Session state.	ClientRegionShortcut.PROXY	EnableGemFireHttpSession.clientRegionShortcu
cache.server.region.shortcut	Configures the DataPolicy used by the server Region to manage (HTTP) Session state.	RegionShortcut.PARTITION	EnableGemFireHttpSession.serverRegionShortc
session.attributes.indexable	Configures names of Session attributes for which an Index will be created.	[]	EnableGemFireHttpSession.indexableSessionAt
session.expiration.max-inactive-interval-seconds	Configures the number of seconds in which a Session can remain inactive before it expires.	1800	EnableGemFireHttpSession.maxInactiveIntervalS
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 org.springframework.session.data.gemfire.serialization.SessionSerializer.		EnableGemFireHttpSession.sessionSerializerBe

Apache Geode Properties

While 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.

> **Warning**
>
> Apache Geode (and Pivotal GemFire) are very strict about the properties that maybe specified in a `gemfire.properties` file. You cannot mix Spring properties with `gemfire.*` properties in either a Spring Boot `application.properties` file or an Apache Geode `gemfire.properties` file.

19.3 Disabling Auto-configuration

If you would like to disable the *auto-configuration* of any feature provided by Spring Boot for Apache Geode or Pivotal GemFire, 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 & Pivotal GemFire (SBDG) include:

- `CacheNameAutoConfiguration`
- `CachingProviderAutoConfiguration`
- `ClientCacheAutoConfiguration`
- `ClientSecurityAutoConfiguration`
- `ContinuousQueryAutoConfiguration`
- `FunctionExecutionAutoConfiguration`
- `GemFirePropertiesAutoConfiguration`
- `LoggingAutoConfiguration`
- `PdxSerializationAutoConfiguration`
- `PeerSecurityAutoConfiguration`
- `RegionTemplateAutoConfiguration`
- `RepositoriesAutoConfiguration`
- `SpringSessionAutoConfiguration`

- `SpringSessionPropertiesAutoConfiguration`
- `SslAutoConfiguration`

19.4 Switch from Apache Geode to Pivotal Cloud Cache (a.k.a. Pivotal GemFire)

First, understand that Pivotal GemFire is being succeeded by Pivotal Cloud Cache (PCC). Therefore, all references to Pivotal GemFire implies for Pivotal Cloud Cache (PCC) as well.

When it comes to Spring's support, whether you are developing with Open Source Software (OSS) Apache Geode or developing for Pivotal Cloud Cache, Spring has you covered.

At a strategic-level, this means:

1. From *Open Source Software* (e.g. Apache Geode) to *Commercial* (e.g. Pivotal Cloud Cache)
2. From *Non-Managed Environments* (e.g. Standalone, Externally Managed) to *Managed Environments* (e.g. Pivotal Platform)
3. With ***little*** to ***no code or configuration*** changes necessary. It just works!

You may also migrate your Spring Boot applications away from Pivotal Platform using the commercial software offering, Pivotal Cloud Cache, and switch back to Open Source Apache Geode running in a standalone, externally managed environment.

SBDG will never lock you in! It is, and always will be, your choice!

Technically, this means to go from Apache Geode to Pivotal Cloud Cache (PCC), you must change 2 things.

First, you must switch the dependency from `spring-geode-starter` to `spring-gemfire-starter`:

Maven POM with Spring Boot for Apache Geode.

```
<dependency>
    <groupId>org.springframework.geode</groupId>
    <artifactId>spring-geode-starter</artifactId>
    <version>1.2.10.RELEASE</version>
</dependency>
```

Gradle build file with Spring Boot for Apache Geode.

```
dependencies {
    compile 'org.springframework.geode:spring-geode-starter:1.2.10.RELEASE'
}
```

To:

Maven POM with Spring Boot for Pivotal GemFire.

```
<dependency>
    <groupId>org.springframework.geode</groupId>
    <artifactId>spring-gemfire-starter</artifactId>
    <version>1.2.10.RELEASE</version>
</dependency>
```

Gradle build file with Spring Boot for Pivotal GemFire.

```
dependencies {
    compile 'org.springframework.geode:spring-gemfire-starter:1.2.10.RELEASE'
}
```

Second, to obtain the commercial Pivotal Cloud Cache (PCC) or Pivotal GemFire bits, you must declare the appropriate repository declaration in your Maven POM or Gradle build file:

Maven Repository definition to acquire PCC or Pivotal GemFire bits.

```
<repositories>
  <repository>
    <id>pivotal-repository</id>
    <name>Pivotal Commercial Repository</name>
    <url>https://commercial-repo.pivotal.io/data3/gemfire-release-repo/gemfire</url>
  </repository>
</repositories>
```

Gradle Repository definition to acquire PCC or Pivotal GemFire bits.

```
repositories {
  maven { url 'https://commercial-repo.pivotal.io/data3/gemfire-release-repo/gemfire' }
}
```

Accessing the Pivotal Commercial Repository requires you to sign up and authenticate with Pivotal Network (a.k.a. "*PivNet*"). Once you have signed up and successfully created your account, you can use your username and password to configure access to the server in your Maven settings (i.e. ~/.m2/settings.xml):

~/.m2/settings.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.1.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.1.0 https://maven.apache.org/xsd/
settings-1.1.0.xsd">

  <servers>
    <server>
      <id>pivotal-repository</id>
      <username>yourUsername</username>
      <password>yourPassword</password>
    </server>
  </servers>

</settings>
```

> **Tip**
>
> For more details on acquiring the Pivotal Cloud Cache (PCC) or Pivotal GemFire bits, see Pivotal GemFire's documentation. When using the `spring-gemfire-starter`, you do not need to declare the Pivotal GemFire dependencies noted in the documentation. The `spring-gemfire-starter` does that for you! You only need to declare the repository and configure your settings.

To go back, simple change `spring-gemfire-starter` to `spring-geode-starter`.

Done!

It should just work without any code or configuration changes and if this is not the case, for whatever reason, then we will work to correct it, short of any feature differences between Pivotal Cloud Cache (PCC) that cannot be accomplished with Apache Geode itself, of course.

Spring Boot's *auto-configuration* and *convention over configuration* approach tries to determine the runtime environment in order to handle infrastructure logistics so you do not have to. This is true inside or outside of a managed environment so that we can provide users with a consistent and reliable experience without all the hassle and issues that arise by switching environments in the first place. Switching environments is especially common as you migrate your Spring Boot applications from DEV to TEST, followed by STAGING, and finally, to PRODUCTION.

Of course, it will nearly always be easier to "run" Apache Geode as a "managed" service inside Pivotal Platform (PCF) using Pivotal Cloud Cache (PCC) than it will be to manage an externally run Apache Geode cluster, especially if your Use Cases require maximum performance and high availability. We highly recommend this approach when and where possible, but it is still your choice.

19.5 Running an Apache Geode or Pivotal GemFire cluster using Spring Boot from your IDE

As described in Chapter 4, *Building ClientCache Applications*, it is possible to configure and run a small Apache Geode or Pivotal GemFire 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/Pivotal GemFire includes such a class:

Spring Boot application class used to configure and boostrap an Apache Geode/Pivotal GemFire server.

```
@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 or Pivotal GemFire 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 and Pivotal GemFire 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 1 run profile configuration.

```
-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:/Library/Java/JavaVirtualMachines/
jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/cldrdata.jar:/Library/Java/JavaVirtualMachines/
jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/dnsns.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Contents/Home/jre/lib/ext/jaccess.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/
Home/jre/lib/ext/jfxrt.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/
lib/ext/localedata.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/lib/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/Contents/Home/jre/lib/ext/sunpkcs11.jar:/Library/
Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/ext/zipfs.jar:/Library/Java/
JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/javaws.jar:/Library/Java/JavaVirtualMachines/
jdk1.8.0_152.jdk/Contents/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/jre/lib/jsse.jar:/Library/
Java/JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/management-agent.jar:/Library/Java/
JavaVirtualMachines/jdk1.8.0_152.jdk/Contents/Home/jre/lib/plugin.jar:/Library/Java/JavaVirtualMachines/
jdk1.8.0_152.jdk/Contents/Home/jre/lib/resources.jar:/Library/Java/JavaVirtualMachines/jdk1.8.0_152.jdk/
Contents/Home/jre/lib/rt.jar:/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.jar:/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/aspectjweaver-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-
```

You can now connect to this server using *Gfsh*:

Connect with 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

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: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 and Pivotal GemFire require 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-clusted 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 GemFire/Geode `locators` property to "*localhost[10334]*". By default, it assumes the Locator process/service is running on "*locahost*", 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.

> **Tip**
>
> 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`.

> **Tip**
>
> Both the `spring.data.gemfire.name` and `spring.data.gemfire.cache.server.port` properties are well-known properties used by SDG to dynamically configure GemFire/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/Pivotal GemFire cluster, with 3 members, using Spring Boot from inside your IDE.

It is pretty simple to build and run a Spring Boot, Apache Geode/Pivotal GemFire, `ClientCache` application that connects to this cluster. Simply include and use Spring Boot for Apache Geode/Pivotal GemFire, ;-).

19.6 Testing

Spring Test for Apache Geode & Pivotal GemFire is a new, soon to be released and upcoming project to help developers write both *Unit* and *Integration Tests* when using either Apache Geode or Pivotal GemFire in a Spring context.

In fact, the entire test suite in Spring Boot for Apache Geode & Pivotal GemFire is based on this project.

All Spring projects integrating with either Apache Geode or Pivotal GemFire will use this new test framework for all their testing needs, making this new test framework for Apache Geode and Pivotal GemFire a proven and reliable solution for all your Apache Geode/Pivotal GemFire application testing needs when using Spring as well.

Later on, this reference guide will include and dedicate an entire chapter on testing.

19.7 Examples

The definitive source of truth on how to best use Spring Boot for Apache Geode & Pivotal GemFire (or Pivotal Cloud Cache (PCC)) is to refer to the Samples.

Refer to the Pivotal Cloud Cache (PCC), Pizza Store, Spring Boot application for an example of how to use Spring Boot for Pivotal GemFire (SBDG) in a `ClientCache` application interfacing with PCC.

Additionally, you may refer to the Temperature Service, Spring Boot application, which implements 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 & Pivotal GemFire (SDG) as yet another example.

19.8 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
8. Pivotal GemFire User Guide | Javadoc