Spring Boot Auto-configuration for Apache Geode & Pivotal GemFire

John Blum :apache-geode-version: {apache-geode-doc-version} :apache-geode-docs: https://geode.apache.org/docs/guide/{apache-geode-version} :toc: left :toclevels: 2 :stylesdir: ../ :highlightjsdir: ../js/highlight :docinfodir: guides

This guide walks you through building a simple Customer Service, Spring Boot application using Apache Geode to manage Customer interactions. You should already be familiar with Spring Boot and Apache Geode.

By the end of this lesson, you should have a better understanding of what Spring Boot for Apache Geode's (SBDG) *auto-configuration* support actually does.

This guide compliments the Auto-configuration vs. Annotation-based configuration chapter with concrete examples.

Let's begin.

This guide builds on the *Simplifying Apache Geode with Spring Data* presentation by John Blum during the 2017 SpringOne Platform conference. While this example as well as the example presented in the talk both use Spring Boot, only this example is using Spring Boot for Apache Geode (SBDG). This guide improves on the example from the presentation by using SBDG.

link:https://docs.spring.io/spring-boot-data-geode-build/current/reference/html5/guides/boot-configuration.html

link:https://docs.spring.io/spring-boot-data-geode-build/current/reference/html5/index.html\#geode-samples

Application Domain Classes

We will build the Spring Boot, Customer Service application from the ground up.

`Customer` class

Like any sensible application development project, we begin by modeling the data our application needs to manage, namely a `Customer`. For this example, the `Customer` class is implemented as follows:

Customer class.

```
@Region("Customers")
@EqualsAndHashCode
@ToString(of = "name")
@RequiredArgsConstructor(staticName = "newCustomer")
public class Customer {

    @Id @NonNull @Getter
    private Long id;

    @NonNull @Getter
    private String name;

}
```

The `Customer` class uses Project Lombok to simplify the implementation so we can focus on the details we care about. Lombok is useful for testing or prototyping purposes. However, using Project Lombok is optional and in most production applications, and I would not recommend it.

Additionally, the `Customer` class is annotated with Spring Data Geode's (SDG) `@Region` annotation. `@Region` is a mapping annotation declaring the Apache Geode cache `Region` in which `Customer` data will be persisted.

Finally, the `org.springframework.data.annotation.Id` annotation was used to designate the `Customer.id` field as the identifier for `Customer` objects. The identifier is the Key used in the Entry stored in the "Customers"`Region`. A `Region` is a distributed version of `java.util.Map`.

If the `@Region` annotation is not explicitly declared, then SDG uses the simple name of the class, which in this case is "Customer", to identify the `Region`. However, there is another reason we explicitly annotated the `Customer` class with `@Region`, which we will cover below.

`CustomerRepository` interface

Next, we create a *Data Access Object* (DAO) to persist `Customers` to Apache Geode. We create the DAO using Spring Data's *Repository* abstraction:

CustomerRepository inteface.

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

    Customer findByNameLike(String name);

}
```

`CustomerRepository` is a Spring Data `CrudRepository`. `CrudRepository` provides basic CRUD (CREATE, READ, UPDATE, and DELETE) data access operations along with the ability to define simple queries on `Customers`.

Spring Data Geode will create a proxy implementation for your application-specific *Repository* interfaces, implementing any query methods you may have explicitly defined on the interface in addition to the data access operations provided in the `CrudRepository` interface extension.

In addition to the base `CrudRepository` operations, `CustomerRepository` has additionally defined a `findByNameLike(:String):Customer` query method. The Apache Geode OQL query is derived from the method declaration.

Though it is beyond the scope of this document, Spring Data's *Repository* infrastructure is capable of generating data store specific queries (e.g. Apache Geode OQL) for *Repository* interface query method declarations just by introspecting the method signature. The query methods must conform to specific conventions. Alternatively, users may use `@Query` to annotate query methods to specify the raw query instead (i.e. OQL for Apache Geode, SQL for JDBC, possibly HQL for JPA, and so on).

`CustomerServiceApplication` (Spring Boot main class)

Now that we have created the basic domain classes of our Customer Service application, we need a main application class to drive the interactions with Customers:

CustomerServiceApplication class.

```
@SpringBootApplication
@EnableClusterAware
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
public class CustomerServiceApplication {

    public static void main(String[] args) {

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

    @Bean
    ApplicationRunner runner(CustomerRepository customerRepository) {

        return args -> {

            assertThat(customerRepository.count()).isEqualTo(0);

            Customer jonDoe = Customer.newCustomer(1L, "Jon Doe");

            System.err.printf("Saving Customer [%s]%n", jonDoe);

            jonDoe = customerRepository.save(jonDoe);

            assertThat(jonDoe).isNotNull();
            assertThat(jonDoe.getId()).isEqualTo(1L);
            assertThat(jonDoe.getName()).isEqualTo("Jon Doe");
            assertThat(customerRepository.count()).isEqualTo(1);

            System.err.println("Querying for Customer [SELECT * FROM /Customers WHERE name LIKE '%Doe']");

            Customer queriedJonDoe = customerRepository.findByNameLike("%Doe");
```

```
            assertThat(queriedJonDoe).isEqualTo(jonDoe);

            System.err.printf("Customer was [%s]%n", queriedJonDoe);
        };
    }
}
```

The `CustomerServiceApplication` class is annotated with `@SpringBootApplication`. Therefore, the main class is a proper Spring Boot application equipped with all the features of Spring Boot (e.g. *auto-configuration*).

Additionally, we use Spring Boot's `SpringApplicationBuilder` in the `main` method to configure and bootstrap the Customer Service application.

Then, we declare a Spring Boot `ApplicationRunner` bean, which is invoked by Spring Boot after the Spring container (i.e. `ApplicationContext`) has been properly initialized and started. Our `ApplicationRunner` defines the Customer interactions performed by our Customer Service application.

Specifically, the runner creates a new `Customer` object ("Jon Doe"), saves him to the "Customers" Region, and then queries for "Jon Doe" using an OQL query with the predicate: `name LIKE '%Doe'`.

`%` is the wildcard for OQL text searches.

Running the Example

You can run the `CustomerServiceApplication` class from your IDE (e.g. IntelliJ IDEA) or from the command-line with the `gradlew` command.

There is nothing special you must do to run the `CustomerServiceApplication` class from inside your IDE. Simply create a run profile configuration and run it.

There is also nothing special about running the `CustomerServiceApplication` class from the command-line using `gradlew`. Simply execute it with `bootRun`:

```
$ gradlew :spring-geode-samples-boot-configuration:bootRun
```

If you wish to adjust the log levels for either Apache Geode or Spring Boot while running the example, then you can set the log level for the individual Loggers (i.e. `org.apache` or `org.springframework`) in `src/main/resources/logback.xml`:

spring-geode-samples/boot/configuration/src/main/resources/logback.xml.

```
include::/Users/wlund/Dropbox/git-workspace/wxlund/spring-geode-workshop/configuration/src/main/resources/lo
```

Auto-configuration for Apache Geode, Take One

"*With great power comes great responsibility.*" - Uncle Ben

While it is not apparent (yet), there is a lot of hidden, intrinsic power provided by Spring Boot Data Geode (SBDG) in this example.

Cache instance

First, in order to put anything into Apache Geode you need a cache instance. A cache instance is also required to create `Regions` which ultimately store the application's data (state). Again, a `Region` is just a Key/Value data structure, like `java.util.Map`, mapping a Key to a Value, or an Object. A `Region` is actually much more than a simple `Map` since it is distributed. However, since `Region` implements `java.util.Map`, it can be treated as such.

A complete discussion of `Region` and it concepts are beyond the scope of this document. You may learn more by reading Apache Geode's User Guide on {apache-geode-docs}/developing/region_options/chapter_overview.html[Regions].

SBDG is opinionated and assumes most Apache Geode applications will be client applications in Apache Geode's {apache-geode-docs}/topologies_and_comm/cs_configuration/chapter_overview.html[client/server topology]. Therefore, SBDG auto-configures a `ClientCache` instance by default.

The intrinsic `ClientCache` *auto-configuration* provided by SBDG can be made apparent by disabling it:

Disabling ClientCache Auto-configuration.

```
@SpringBootApplication(exclude = ClientCacheAutoConfiguration.class)
@EnableEntityDefinedRegions(basePackageClasses = Customer.class, clientRegionShortcut = ClientRegionShortcut
public class CustomerServiceApplication {
  // ...
}
```

Note the `exclude` on the `ClientCacheAutoConfiguration.class`.

With the correct log level set, you will see an error message similar to:

Error resulting from no ClientCache instance.

```
16:20:47.543 [main] DEBUG o.s.b.d.LoggingFailureAnalysisReporter - Application failed to start due to an exce
org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'example.app.crm
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.raiseNoMatchingBeanFound(Default
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultLista
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListable
    at org.springframework.beans.factory.support.ConstructorResolver.resolveAutowiredArgument(ConstructorReso
    ...
16:20:47.548 [main] ERROR o.s.b.d.LoggingFailureAnalysisReporter -

***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of method runner in example.app.crm.CustomerServiceApplication required a bean of type 'example.a
```

Essentially, the `CustomerRepository` could not be injected into our `CustomerServiceApplication` class, `ApplicationRunner` bean method because the `CustomerRepository`, which depends on the "Customers" Region, could not be created. The `CustomerRepository` could not be created because the "Customers" Region could not be created. The "Customers" Region could not be created because there was no cache instance available (e.g. `ClientCache`) to create `Regions`, resulting in a trickling effect.

The `ClientCache` *auto-configuration* is equivalent to the following:

Equivalent ClientCache configuration.

```
@SpringBootApplication
@ClientCacheApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class, clientRegionShortcut = ClientRegionShortcut
public class CustomerServiceApplication {
  // ...
}
```

That is, you would need to explicitly declare the `@ClientCacheApplication` annotation if you were not using SBDG.

Repository instance

We are also using the Spring Data (Geode) *Repository* infrastructure in the Customer Service application. This should be evident from our declaration and definition of the application-specific `CustomerRepository` interface.

If we disable the Spring Data *Repository auto-configuration*:

Disabling Spring Data Repositories Auto-configuration.

```
@SpringBootApplication(exclude = RepositoriesAutoConfiguration.class)
@EnableEntityDefinedRegions(basePackageClasses = Customer.class, clientRegionShortcut = ClientRegionShortcut
public class CustomerServiceApplication {
  // ...
}
```

The application would throw a similar error on startup:

Error resulting from no proxied `CustomerRepository` instance.

```
17:31:21.231 [main] DEBUG o.s.b.d.LoggingFailureAnalysisReporter - Application failed to start due to an exce
org.springframework.beans.factory.NoSuchBeanDefinitionException: No qualifying bean of type 'example.app.crm
```

```
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.raiseNoMatchingBeanFound(Default
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.doResolveDependency(DefaultLista
    at org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListable
    at org.springframework.beans.factory.support.ConstructorResolver.resolveAutowiredArgument(ConstructorRes
    ...
17:31:21.235 [main] ERROR o.s.b.d.LoggingFailureAnalysisReporter -

***************************
APPLICATION FAILED TO START
***************************

Description:

Parameter 0 of method runner in example.app.crm.CustomerServiceApplication required a bean of type 'example.a
```

In this case, there was simply no proxy implementation for the `CustomerRepository` interface provided by the framework since the *auto-configuration* was disabled. The `ClientCache` and "Customers" `Region` do exist in this case, though.

The Spring Data *Repository auto-configuration* even takes care of locating our application *Repository* interface definitions for us.

Without *auto-configuration*, you would need to explicitly:

Equivalent Spring Data Repositories configuration.

```
@SpringBootApplication(exclude = RepositoriesAutoConfiguration.class)
@EnableEntityDefinedRegions(basePackageClasses = Customer.class, clientRegionShortcut = ClientRegionShortcut
@EnableGemfireRepositories(basePackageClasses = CustomerRepository.class)
public class CustomerServiceApplication {
  // ...
}
```

That is, you would need to explicitly declare the `@EnableGemfireRepositories` annotation and set the `basePackages` attribute, or the equivalent, type-safe `basePackageClasses` attribute, to the package containing your application *Repository* interfaces, if you were not using SBDG.

Entity-defined Regions

So far, the only explicit declaration of configuration in our Customer Service application is the `@EnableEntityDefinedRegions` annotation.

As was alluded to above, there was another reason we explicitly declared the `@Region` annotation on our `Customer` class.

We could have defined the client `LOCAL` "Customers" Region using Spring JavaConfig, explicitly:

JavaConfig Bean Definition for the "Customers" Region.

```
@Configuration
class ApplicationConfiguration {

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

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

        customersRegion.setCache(gemfireCache);
        customersRegion.setShortcut(ClientRegionShortcut.LOCAL);

        return customersRegion;
    }
}
```

Or, even define the "Customers" Region using Spring XML, explicitly:

XML Bean Definition for the "Customers" Region.

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

But, using SDG's `@EnableEntityDefinedRegions` annotation is very convenient and can scan for the Regions (whether client or server (peer) Regions) required by your application based the entity classes themselves (e.g. `Customer`):

Annotation-based config for the "Customers" Region.

```
@EnableEntityDefinedRegions(basePackageClasses = Customer.class, clientRegionShortcut = ClientRegionShortcut
class CustomerServiceApplication { }
```

The `basePackageClasses` attribute is an alternative to `basePackages`, and a type-safe way to target the packages (and subpackages) containing the entity classes that your application will persist to Apache Geode. You only need to choose one class from each top-level package for where you want the scan to begin. Spring Data Geode uses this class to determine the package to begin the scan. 'basePackageClasses` accepts an array of `Class` types so you can specify multiple independent top-level packages. The annotation also includes the ability to filter types.

However, the `@EnableEntityDefinedRegions` annotation only works when the entity class (e.g. `Customer`) is explicitly annotated with the `@Region` annotation (e.g. `@Region("Customers")`), otherwise it ignores the class.

You will also notice that the data policy type (i.e. `clientRegionShort`, or simply `shortcut`) is set to `LOCAL` in our example. Why?

Well, initially we just want to get up and running as quickly as possible, without a lot of ceremony and fuss. By using a client `LOCAL` Region to begin with, we are not required to start a cluster of servers for the client to be able to store data.

While client `LOCAL` Regions can be useful for some purposes (e.g. local processing, querying and aggregating of data), it is more common for a client to persist data in a cluster of servers, and for that data to be shared by multiple clients (instances) in the application architecture, especially as the application is scaled out to handle demand.

Switching to Client/Server

We continue with our example by switching from a local context to a client/server topology.

If you are rapidly prototyping and developing your application and simply want to lift off the ground quickly, then it is useful to start locally and gradually migrate towards a client/server architecture.

To switch to client/server, all you need to do is remove the `clientRegionShortcut` attribute configuration from the `@EnableEntityDefinedRegions` annotation declaration:

Client/Server Topology Region Configuration.

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

The default value for the `clientRegionShortcut` attribute is `ClientRegionShortcut.PROXY`. This means no data is stored locally. All data is sent from the client to one or more servers in a cluster.

However, if we try to run the application, it will fail:

NoAvailableServersException.

```
Caused by: org.apache.geode.cache.client.NoAvailableServersException
    at org.apache.geode.cache.client.internal.pooling.ConnectionManagerImpl.borrowConnection(ConnectionManage
    at org.apache.geode.cache.client.internal.OpExecutorImpl.execute(OpExecutorImpl.java:136) ~[geode-core-1
    at org.apache.geode.cache.client.internal.OpExecutorImpl.execute(OpExecutorImpl.java:115) ~[geode-core-1
    at org.apache.geode.cache.client.internal.PoolImpl.execute(PoolImpl.java:763) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.cache.client.internal.QueryOp.execute(QueryOp.java:58) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.cache.client.internal.ServerProxy.query(ServerProxy.java:70) ~[geode-core-1.2.1.jar:
    at org.apache.geode.cache.query.internal.DefaultQuery.executeOnServer(DefaultQuery.java:456) ~[geode-core
    at org.apache.geode.cache.query.internal.DefaultQuery.execute(DefaultQuery.java:338) ~[geode-core-1.2.1.
    at org.springframework.data.gemfire.GemfireTemplate.find(GemfireTemplate.java:311) ~[spring-data-geode-2
    at org.springframework.data.gemfire.repository.support.SimpleGemfireRepository.count(SimpleGemfireReposit
    ...
    at example.app.crm.CustomerServiceApplication.lambda$runner$0(CustomerServiceApplication.java:59) ~[class
    at org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:783) ~[spring-boot-2.0.9
```

The client is expecting there to be a cluster of servers to communicate with and to store/access data from. Clearly, there are no servers or cluster running yet.

There are several ways in which to start a cluster. For example, you may use Spring to configure and bootstrap the cluster, which has been demonstrated here.

Although, for this example, we are going to use the tools provided with Apache Geode, or Pivotal GemFire, i.e. *Gfsh* (GemFire/Geode Shell) for reasons that will become apparent later.

You need to download and install a full distribution of Apache Geode to make use of the provided tools. After installation, you will need to set the `GEODE` (or `GEMFIRE`) environment variable to the location of your installation. Additionally, add `$GEODE/bin` to your system `$PATH`.

Once Apache Geode has been successfully installed, you can open a command prompt (terminal) and do:

Running Gfsh.

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


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

Monitor and Manage Apache Geode
gfsh>
```

You are set to go.

For your convenience, a *Gfsh* shell script is provided to start a cluster:

Gfsh shell script.

```
# Gfsh shell script to start a simple GemFire/Geode cluster

start locator --name=LocatorOne --log-level=config
configure pdx --read-serialized=true
start server --name=ServerOne --log-level=config
```

Specifically, we are starting 1 Locator and 1 Server, all running with the default ports.

Execute the *Gfsh* shell script using:

Run Gfsh shell script.

```
gfsh>run --file=/path/to/spring-boot-data-geode/samples/boot/configuration/src/main/resources/geode/bin/start
1. Executing - start locator --name=LocatorOne --log-level=config

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: 68425
Uptime: 2 seconds
Geode Version: 1.2.1
Java Version: 1.8.0_192
Log File: /Users/jblum/pivdev/lab/LocatorOne/LocatorOne.log
JVM Arguments: -Dgemfire.log-level=config -Dgemfire.enable-cluster-configuration=true -Dgemfire.load-cluster-
Class-Path: /Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-core-1.2.1.jar:/Users/jblum/pivdev/apache-geode-

Successfully connected to: JMX Manager [host=10.99.199.24, port=1099]

Cluster configuration service is up and running.

2. Executing - start server --name=ServerOne --log-level=config

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: 68434
Uptime: 2 seconds
Geode Version: 1.2.1
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.use-cluster-configuration=true -Dgem
Class-Path: /Users/jblum/pivdev/apache-geode-1.2.1/lib/geode-core-1.2.1.jar:/Users/jblum/pivdev/apache-geode-
```

You will need to change the path to the `spring-boot-data-geode/samples/boot/configuration` directory in the `run --file=…` *Gfsh* command above based on where you git cloned the `spring-boot-data-geode` project to your computer.

Now, our simple cluster with an Apache Geode Locator and (Cache) Server is running. We can verify by listing and describing the members:

List and Describe Members.

```
gfsh>list members
   Name     | Id
----------- | ----------------------------------------------------
LocatorOne | 10.99.199.24(LocatorOne:68425:locator)<ec><v0>:1024
ServerOne  | 10.99.199.24(ServerOne:68434)<v1>:1025


gfsh>describe member --name=ServerOne
Name        : ServerOne
Id          : 10.99.199.24(ServerOne:68434)<v1>:1025
Host        : 10.99.199.24
Regions     :
PID         : 68434
Groups      :
Used Heap   : 27M
Max Heap    : 3641M
Working Dir : /Users/jblum/pivdev/lab/ServerOne
Log file    : /Users/jblum/pivdev/lab/ServerOne/ServerOne.log
Locators    : 10.99.199.24[10334]

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

What happens if we try to run the application now?

RegionNotFoundException.

```
17:42:16.873 [main] ERROR o.s.b.SpringApplication - Application run failed
java.lang.IllegalStateException: Failed to execute ApplicationRunner
    ...
    at example.app.crm.CustomerServiceApplication.main(CustomerServiceApplication.java:51) [classes/:?]
Caused by: org.springframework.dao.DataAccessResourceFailureException: remote server on 10.99.199.24(SpringBa
    at org.springframework.data.gemfire.GemfireCacheUtils.convertGemfireAccessException(GemfireCacheUtils.jav
    at org.springframework.data.gemfire.GemfireAccessor.convertGemFireAccessException(GemfireAccessor.java:91
    at org.springframework.data.gemfire.GemfireTemplate.find(GemfireTemplate.java:329) ~[spring-data-geode-2
    at org.springframework.data.gemfire.repository.support.SimpleGemfireRepository.count(SimpleGemfireReposit
    ...
    at example.app.crm.CustomerServiceApplication.lambda$runner$0(CustomerServiceApplication.java:59) ~[class
    at org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:783) ~[spring-boot-2.0.9
    ... 3 more
Caused by: org.apache.geode.cache.client.ServerOperationException: remote server on 10.99.199.24(SpringBased
    at org.apache.geode.cache.client.internal.AbstractOp.processChunkedResponse(AbstractOp.java:352) ~[geode-
    at org.apache.geode.cache.client.internal.QueryOp$QueryOpImpl.processResponse(QueryOp.java:170) ~[geode-c
    at org.apache.geode.cache.client.internal.AbstractOp.processResponse(AbstractOp.java:230) ~[geode-core-1
    at org.apache.geode.cache.client.internal.AbstractOp.attempt(AbstractOp.java:394) ~[geode-core-1.2.1.jar
    at org.apache.geode.cache.client.internal.AbstractOp.attemptReadResponse(AbstractOp.java:203) ~[geode-cor
    at org.apache.geode.cache.client.internal.ConnectionImpl.execute(ConnectionImpl.java:275) ~[geode-core-1
    at org.apache.geode.cache.client.internal.pooling.PooledConnection.execute(PooledConnection.java:332) ~[g
    at org.apache.geode.cache.client.internal.OpExecutorImpl.executeWithPossibleReAuthentication(OpExecutorIm
    at org.apache.geode.cache.client.internal.OpExecutorImpl.execute(OpExecutorImpl.java:158) ~[geode-core-1
    at org.apache.geode.cache.client.internal.OpExecutorImpl.execute(OpExecutorImpl.java:115) ~[geode-core-1
    at org.apache.geode.cache.client.internal.PoolImpl.execute(PoolImpl.java:763) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.cache.client.internal.QueryOp.execute(QueryOp.java:58) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.cache.client.internal.ServerProxy.query(ServerProxy.java:70) ~[geode-core-1.2.1.jar:?
    at org.apache.geode.cache.query.internal.DefaultQuery.executeOnServer(DefaultQuery.java:456) ~[geode-core
    at org.apache.geode.cache.query.internal.DefaultQuery.execute(DefaultQuery.java:338) ~[geode-core-1.2.1.j
    at org.springframework.data.gemfire.GemfireTemplate.find(GemfireTemplate.java:311) ~[spring-data-geode-2
    at org.springframework.data.gemfire.repository.support.SimpleGemfireRepository.count(SimpleGemfireReposit
```

```
    ...
    at example.app.crm.CustomerServiceApplication.lambda$runner$0(CustomerServiceApplication.java:59) ~[class
    at org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:783) ~[spring-boot-2.0.9
    ... 3 more
Caused by: org.apache.geode.cache.query.RegionNotFoundException: Region not found:  /Customers
    at org.apache.geode.cache.query.internal.DefaultQuery.checkQueryOnPR(DefaultQuery.java:599) ~[geode-core-
    at org.apache.geode.cache.query.internal.DefaultQuery.execute(DefaultQuery.java:348) ~[geode-core-1.2.1.
    at org.apache.geode.cache.query.internal.DefaultQuery.execute(DefaultQuery.java:319) ~[geode-core-1.2.1.
    at org.apache.geode.internal.cache.tier.sockets.BaseCommandQuery.processQueryUsingParams(BaseCommandQuery
    at org.apache.geode.internal.cache.tier.sockets.BaseCommandQuery.processQuery(BaseCommandQuery.java:65) ~
    at org.apache.geode.internal.cache.tier.sockets.command.Query.cmdExecute(Query.java:91) ~[geode-core-1.2
    at org.apache.geode.internal.cache.tier.sockets.BaseCommand.execute(BaseCommand.java:165) ~[geode-core-1
    at org.apache.geode.internal.cache.tier.sockets.ServerConnection.doNormalMsg(ServerConnection.java:791) ~
    at org.apache.geode.internal.cache.tier.sockets.ServerConnection.doOneMessage(ServerConnection.java:922)
    at org.apache.geode.internal.cache.tier.sockets.ServerConnection.run(ServerConnection.java:1180) ~[geode-
    ...
```

The application fails to run because we (deliberately) did not create a corresponding, server-side, "Customers" Region. In order for a client to send data via a client `PROXY` Region (a Region with no local state) to a server in a cluster, at least one server in the cluster must have a matching Region by name (i.e. "Customers").

Indeed, there are no Regions in the cluster:

List Regions.

```
gfsh>list regions
No Regions Found
```

Of course, you could create the matching server-side, "Customers" Region using *Gfsh*:

```
gfsh>create region --name=Customers --type=PARTITION
```

But, what if you have hundreds of application domain objects each requiring a Region for persistence? It is not an unusual or unreasonable requirement in any practical enterprise scale application.

While it is not a "convention" in Spring Boot for Apache Geode (SBDG), Spring Data for Apache Geode (SDG) comes to our rescue. We simply only need to enable cluster configuration from the client:

Enable Cluster Configuration.

```
@SpringBootApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableClusterConfiguration(useHttp = true)
public class CustomerServiceApplication {
  // ...
}
```

That is, we additionally annotate our Customer Service application class with SDG's `@EnableClusterConfiguration` annotation. We have also set the `useHttp` attribute to `true`. This sends the configuration metadata from the client to the cluster via GemFire/Geode's Management REST API.

This is useful when your GemFire/Geode cluster may be running behind a firewall, such as on public cloud infrastructure. However, there are other benefits to using HTTP as well. As stated, the client sends configuration metadata to GemFire/Geode's Management REST interface, which is a facade for the server-side Cluster Configuration Service. If another peer (e.g. server) is added to the cluster as a member, then this member will get the same configuration. If the entire cluster goes down, it will have the same configuration when it is restarted.

SDG is careful not to stomp on existing Regions since those Regions may have data already. Declaring the `@EnableClusterConfiguration` annotation is a useful development-time feature, but it is recommended that you explicitly define and declare your Regions in production environments, either using *Gfsh* or Spring confg.

It is now possible to replace the SDG `@EnableClusterConfiguration` annotation with SBDG's `@EnableClusterAware` annotation, which has the same effect of pushing configuration metadata from the client to the server (or cluster). Additionally, SBDG's `@EnableClusterAware` annotation makes it unnecessary to explicitly have to configure the `clientRegionShortcut` on the SDG `@EnableEntityDefinedRegions` annotation (or similar annotation, e.g. SDG's `@EnableCachingDefinedRegions`). Finally, because the SBDG `@EnableClusterAware` annotation is meta-annotated with SDG's `@EnableClusterConfiguration annotation` is automatically configures the `useHttp` attribute to `true`.

Now, we can run our application again, and this time, it works!

Client/Server Run Successful.

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.9.RELEASE)

Saving Customer [Customer(name=Jon Doe)]
Querying for Customer [SELECT * FROM /Customers WHERE name LIKE '%Doe']
Customer was [Customer(name=Jon Doe)]

Process finished with exit code 0
```

In the cluster (server-side), we will also see that the "Customers" Region was created successfully:

List & Describe Regions.

```
gfsh>list regions
List of regions
---------------
Customers


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

Non-Default Attributes Shared By Hosting Members

 Type  |    Name     | Value
------ | ----------- | ---------
Region | size        | 1
       | data-policy | PARTITION
```

We see that the "Customers" Region has a size of 1, containing "Jon Doe".

We can verify this by querying the "Customers" Region:

Query for all Customers.

```
gfsh>query --query="SELECT customer.name FROM /Customers customer"
Result : true
Limit  : 100
Rows   : 1

Result
-------
Jon Doe
```

That was easy!

Auto-configuration for Apache Geode, Take Two

What may not be apparent in this example up to this point is how the data got from the client to the server. Certainly, our client did send `Jon Doe` to the server, but our `Customer` class is not `java.io.Serializable`. So, how was an instance of `Customer` streamed and sent from the client to the server then (it is using a Socket)?

Any object sent over a network, between two Java processes, or streamed to/from disk, must be serializable, no exceptions!

Furthermore, when we started the cluster, we did not include any application domain classes on the classpath of any server in the cluster.

As further evidence, we an adjust our query slightly:

Invalid Query.

```
gfsh>query --query="SELECT * FROM /Customers"
Message : Could not create an instance of a class example.app.crm.model.Customer
Result  : false
```

If you tried to perform a `get`, you would hit a similar error:

Region.get(key).

```
gfsh>get --region=/Customers --key=1 --key-class=java.lang.Long
Message : Could not create an instance of a class example.app.crm.model.Customer
Result  : false
```

So, how was the data sent then? How were we able to access the data stored in the server(s) on the cluster with the OQL query `SELECT customer.name FROM /Customers customer` as seen above?

Well, Apache Geode and Pivotal GemFire provide 2 proprietary serialization formats in addition to *Java Serialization*: {apache-geode-docs}/developing/data_serialization/gemfire_data_serialization.html[Data Serialization] and {apache-geode-docs}/developing/data_serialization/gemfire_pdx_serialization.html[PDX], or *Portable Data Exchange*.

While *Data Serialization* is more efficient, PDX is more flexible (i.e. "portable"). PDX enables data to be queried in serialized form and is the format used to support both Java and Native Clients (C++, C#) simultaneously. Therefore, PDX is auto-configured in Spring Boot Data Geode (SBDG) by default.

This is convenient since you may not want to implement `java.io.Serializable` for all your application domain model types that you store in Apache Geode. In other cases, you may not even have control over the types referred to by your application domain model types to make them `Serializable`, such as when using a 3rd party library.

So, SBDG auto-configures PDX and uses Spring Data Geode's `MappingPdxSerializer` as the `PdxSerializer` to de/serialize all application domain model types.

If we disable PDX *auto-configuration*, we will see the effects of trying to serialize a non-serializable type, `Customer`.

First, let's back up a few steps and destroy the server-side "Customers" Region:

Destroy "Customers" Region.

```
gfsh>destroy region --name=/Customers
"/Customers"  destroyed successfully.


gfsh>list regions
No Regions Found
```

Then, we disable PDX *auto-configuration*:

Disable PDX Auto-configuration.

```
@SpringBootApplication(exclude = PdxSerializationAutoConfiguration.class)
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableClusterConfiguration(useHttp = true)
public class CustomerServiceApplication {
  // ...
}
```

When we re-run the application, we get the error we would expect:

NotSerializableException.

```
Caused by: java.io.NotSerializableException: example.app.crm.model.Customer
    at java.io.ObjectOutputStream.writeObject0(ObjectOutputStream.java:1184) ~[?:1.8.0_192]
    at java.io.ObjectOutputStream.writeObject(ObjectOutputStream.java:348) ~[?:1.8.0_192]
    at org.apache.geode.internal.InternalDataSerializer.writeSerializableObject(InternalDataSerializer.java:
    at org.apache.geode.internal.InternalDataSerializer.basicWriteObject(InternalDataSerializer.java:2123) ~
    at org.apache.geode.DataSerializer.writeObject(DataSerializer.java:2936) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.internal.util.BlobHelper.serializeTo(BlobHelper.java:66) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.internal.cache.tier.sockets.Message.serializeAndAddPart(Message.java:396) ~[geode-co
    at org.apache.geode.internal.cache.tier.sockets.Message.addObjPart(Message.java:340) ~[geode-core-1.2.1.
    at org.apache.geode.internal.cache.tier.sockets.Message.addObjPart(Message.java:319) ~[geode-core-1.2.1.
    at org.apache.geode.cache.client.internal.PutOp$PutOpImpl.<init>(PutOp.java:281) ~[geode-core-1.2.1.jar:
    at org.apache.geode.cache.client.internal.PutOp.execute(PutOp.java:66) ~[geode-core-1.2.1.jar:?]
```

```
    at org.apache.geode.cache.client.internal.ServerRegionProxy.put(ServerRegionProxy.java:162) ~[geode-core-
    at org.apache.geode.internal.cache.LocalRegion.serverPut(LocalRegion.java:3006) ~[geode-core-1.2.1.jar:?
    at org.apache.geode.internal.cache.LocalRegion.cacheWriteBeforePut(LocalRegion.java:3115) ~[geode-core-1
    at org.apache.geode.internal.cache.ProxyRegionMap.basicPut(ProxyRegionMap.java:222) ~[geode-core-1.2.1.ja
    at org.apache.geode.internal.cache.LocalRegion.virtualPut(LocalRegion.java:5628) ~[geode-core-1.2.1.jar:
    at org.apache.geode.internal.cache.LocalRegionDataView.putEntry(LocalRegionDataView.java:151) ~[geode-co
    at org.apache.geode.internal.cache.LocalRegion.basicPut(LocalRegion.java:5057) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.internal.cache.LocalRegion.validatedPut(LocalRegion.java:1595) ~[geode-core-1.2.1.ja
    at org.apache.geode.internal.cache.LocalRegion.put(LocalRegion.java:1582) ~[geode-core-1.2.1.jar:?]
    at org.apache.geode.internal.cache.AbstractRegion.put(AbstractRegion.java:325) ~[geode-core-1.2.1.jar:?]
    at org.springframework.data.gemfire.GemfireTemplate.put(GemfireTemplate.java:193) ~[spring-data-geode-2.
    at org.springframework.data.gemfire.repository.support.SimpleGemfireRepository.save(SimpleGemfireReposit
    ...
    at example.app.crm.CustomerServiceApplication.lambda$runner$0(CustomerServiceApplication.java:70) ~[spri
    at org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:783) ~[spring-boot-2.0.9
    ...
```

Our "Customers" Region is recreated, but is empty:

Empty "Customers" Region.

```
gfsh>list regions
List of regions
---------------
Customers


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

Non-Default Attributes Shared By Hosting Members

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

So, SBDG takes care of all your serialization needs without you having to configure serialization or implement `java.io.Serializable` in all your application domain model types, including types your application domain model types might refer to, which may not be possible.

If you were not using SBDG, then you would need to enable PDX serialization explicitly.

The PDX *auto-configuration* provided by SBDG is equivalent to:

Equivalent PDX Configuration.

```
@SpringBootApplication
@ClientCacheApplication
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@EnableClusterConfiguration(useHttp = true)
@EnablePdx
public class CustomerServiceApplication {
  // ...
}
```

In addition to the `@ClientCacheApplication` annotation, you would need to annotate the `CustomerServiceApplication` class with SDG's `@EnablePdx` annotation, which is responsible for configuring PDX serialization and registering SDG's `MappingPdxSerializer`.

link:https://docs.spring.io/spring-boot-data-geode-build/current/reference/html5/index.html/#geode-samples