vmware by Broadcom

Spring Boot for Tanzu GemFire

Spring Boot for Tanzu GemFire 2.0

You can find the most up-to-date technical documentation on the VMware by Broadcom website at:

https://techdocs.broadcom.com/

VMware by Broadcom
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

Spring Boot for Tanzu GemFire

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

This project is a continuation and a logical extension to Spring Data for VMware GemFire's Annotation-based configuration model, and the goals set forth in that model:

*To enable application developers to **get up and running** as **quickly**, **reliably**, and as **easily** as possible.*

In fact, Spring Boot for Tanzu GemFire automatically applies *auto-configuration* to several key *use cases* including, but not limited to:

- *Cache-Aside, Inline, and Near*
- *System of Record* (SOR)
- *Transactions*
- *Distributed Computations*
- *Continuous Queries*
- *Data Serialization*
- *Data Initialization*
- *Spring Boot Actuator*
- *HTTP Session state management*

This initial set of documentation is meant to get users up and running quickly with Spring Boot for Tanzu GemFire.

Release Notes

This topic contains the release notes for Spring Boot for VMware Tanzu GemFire.

Spring Boot 3.3 and GemFire 10.1

2.0.3

06 Dec 2024

Fixed issues:

- Stop `spring-boot-3.3-gemfire-actuator` from exporting specific GemFire version

2.0.2

05 Dec 2024

Fixed issues:

- The “subscription-enabled” property is now honored for the “DEFAULT” pool.

Update of underlying Spring Data, Spring Session dependencies:

- Spring Data 3.3 for VMware GemFire 10.1 - 2.0.2
- Spring Session 3.3 for VMware GemFire 10.1 - 2.0.2
- Spring Boot 3.3.6
- Spring Framework - 6.1.15
- Spring Security BOM - 6.3.5
- Spring Data BOM - 2024.0.6
- Spring Session BOM - 3.3.3

2.0.1

21 Oct 2024

Fixed issues:

- Remove dependency on Spring Test Data GemFire library

Update of underlying Spring Data, Spring Session dependencies:

- Spring Data 3.3 for VMware GemFire 10.1 - 2.0.1
- Spring Session 3.3 for VMware GemFire 10.1 - 2.0.1

2.0.0

16 Oct 2024 Initial 2.0 release of Spring Boot 3.3 for GemFire 10.1

Spring Boot for GemFire 2.0 is designed to provide users with a more reliable and streamlined experience by optimizing the platform for scalability. We have thoughtfully retired certain features that previously caused challenges in high-demand environments. Specifically, we have removed the ability to configure and start GemFire clusters. This change allows us to concentrate on enhancing client-side operability, ensuring a more efficient and reliable user experience.

- If you are using the @EnableClusterAware annotation, we have removed the automatic startup of a local GemFire cluster when one isn't found during application startup. Now, if your application doesn't find a cluster, it will fail fast, allowing for quicker detection and resolution of configuration issues.
- If you are using the @EnableClusterAware annotation, we have removed the ability to push region creation from the client to the server, aligning with best practices for production environments and providing better control when iterating on region definitions.

For a complete list of changes and removed annotations, see Upgrading From Version 1.x to 2.x.

Underlying Spring Libraries * spring-boot-3.3-gemfire-10.1 * Spring Boot 3.3.1 * Spring Framework 6.1.10 * Spring Security BOM 6.3.1 * Spring Data 3.3 for VMware GemFire 10.1 - 2.0.0 * Spring Session 3.3 for VMware GemFire 10.1 - 2.0.0

Compatibility and Versions

This topic lists Spring Boot for VMware Tanzu GemFire compatibility and versions.

Spring Boot for Tanzu GemFire provides the convenience of Spring Boot's convention over configuration approach by using auto-configuration with Spring Framework's powerful abstractions and highly consistent programming model to simplify the development of GemFire applications.

Compatibility

Spring Boot for Tanzu GemFire Artifact	Latest Versions	Compatible GemFire Versions	Compatible Spring Boot Versions	Compatible Spring Framework Versions
spring-boot-3.3-gemfire-10.1	2.0.3	10.1.x	3.3.x	6.1.x

Modules

Your application may require more than one module if, for example, you may need (HTTP) Session state management, or you may need to enable Spring Boot Actuator endpoints for GemFire.

You can declare and use any one of the Spring Boot for Tanzu GemFire modules (in addition to the Spring Boot for Tanzu GemFire dependency).

Spring Boot Actuator

Module	Latest Versions
spring-boot-actuator-3.3-gemfire-10.1	2.0.3

Spring Boot Logging

Module	Latest Versions
spring-boot-logging-3.3-gemfire-10.1	2.0.3

Spring Session

Module	Latest Versions
spring-boot-session-3.3-gemfire-10.1	2.0.3

Upgrading From Version 1.x to 2.x

Server Configuration Annotation Removal

With the transition from version 1.x to 2.x, support for configuring and bootstrapping a Tanzu GemFire server using Spring is discontinued. While you can still leverage Spring for your Tanzu GemFire client applications, servers must now be started using alternative methods such as `gfsh`. For instructions for starting and managing GemFire servers, see the Tanzu GemFire documentation.

The following annotations are removed to accommodate this change:

- `@EnableSecurityManager`
- `@EnableSecurityManagerProxy`
- `@UseDistributedSystemId`
- `@UseLocators`
- `@UseMemberName`

Changes to @EnableClusterAware

The behavior of the `@EnableClusterAware` annotation has also changed.

- `@EnableClusterAware` no longer pushes configurations (such as regions) to the cluster. This change is due to the removal of the `@EnableClusterConfiguration` annotation from Spring Data for GemFire.
- Additionally, it no longer switches applications to local-only mode when a cluster cannot be found. Instead, the application will fail if it expects to connect to a cluster but cannot.
- `@EnableClusterAware` still attempts to auto-connect to a cluster in cloud environments.

Annotations Removed in Spring Data for GemFire

The following annotations have been removed from the underlying Spring Data for GemFire library and are no longer available in Spring Boot for GemFire:

- `@CacheServerApplication`
- `@EnableAuth`
- `@EnableAutoRegionLookup`
- `@EnableCacheServer`
- `@EnableCacheServers`

- `@EnableClusterConfiguration`
- `@EnableGatewayReceiver`
- `@EnableGatewaySender`
- `@EnableGatewaySenders`
- `@EnableGemFireAsLastResource`
- `@EnableHttpService`
- `@EnableIndexing`
- `@EnableLocator`
- `@EnableManager`
- `@EnableMcast`
- `@EnableMemcachedServer`
- `@LocatorApplication`
- `@PeerCacheApplication`
- `@GemfireFunction`
- `@RegionData`
- `@EnableGemfireFunctions`
- `@Indexed`
- `@LocalRegion`
- `@PartitionRegion`
- `@FixedPartition`
- `@ReplicateRegion`

Getting Started

This topic explains how to download Spring Boot for VMware Tanzu GemFire and add the libraries to a project.

The Spring Boot for Tanzu GemFire libraries are available from the Broadcom Customer Support Portal.

Prerequisites

1. Log in to the Broadcom Customer Support Portal with your customer credentials. For more information about login requirements, see the Download Broadcom products and software article.
2. Go to the VMware Tanzu GemFire downloads page, select **VMware Tanzu GemFire**, click **Show All Releases**.
3. Find the release named **Click Green Token for Repository Access** and click the **Token Download** icon on the right. This opens the instructions on how to use the GemFire artifact repository. At the top, the Access Token is provided. Click **Copy to Clipboard**. You will use this Access Token as the password.

Maven

Repository and Credential Setup

1. Modify your project's `pom.xml` file by adding the following repository definition:

    ```
    <repository>
      <id>gemfire-release-repo</id>
      <name>GemFire Release Repository</name>
      <url>https://packages.broadcom.com/artifactory/gemfire/</url>
    </repository>
    ```

2. To access the artifacts, you must add an entry to your `.m2/settings.xml` file:

    ```
    <settings>
      <servers>
        <server>
          <id>gemfire-release-repo</id>
          <username>EXAMPLE-USERNAME</username>
          <password>MY-PASSWORD</password>
        </server>
      </servers>
    </settings>
    ```

    Where:

- `EXAMPLE-USERNAME` is your support.broadcom.com user name.
- `MY-PASSWORD` is the Access Token you copied in step 3 in Prerequisites.

Add the Libraries to your Project

After setting up the repository and credentials, add the Spring Boot for Tanzu GemFire library to your application. To support multiple GemFire versions, the Spring Boot for Tanzu GemFire library requires users to specify an explicit dependency on the desired version of GemFire.

In the following examples:

- Replace the `springBootForGemFire.version` with the version of the library that your project requires.
- Replace the `vmwareGemFire.version` with the version of GemFire that your project requires.
- Replace the `<artifactId>spring-boot-3.3-gemfire-10.1</artifactId>` with the version of Spring Boot + the version of GemFire your application requires.

Add the following to your `pom.xml` file:

```
<properties>
    <springBootForGemFire.version>2.0.3</springBootForGemFire.version>
    <vmwareGemFire.version>10.1.2</vmwareGemFire.version>
</properties>

<dependencies>
   <dependency>
       <groupId>com.vmware.gemfire</groupId>
       <artifactId>spring-boot-3.3-gemfire-10.1</artifactId>
       <version>${springBootForGemFire.version}</version>
   </dependency>
   <dependency>
       <groupId>com.vmware.gemfire</groupId>
       <artifactId>gemfire-core</artifactId>
       <version>${vmwareGemFire.version}</version>
   </dependency>
<!--if using continuous queries-->
   <dependency>
        <groupId>com.vmware.gemfire</groupId>
        <artifactId>gemfire-cq</artifactId>
        <version>${vmwareGemFire.version}</version>
   </dependency>
</dependencies>
```

Gradle

Repository and Credential Setup

1. Add the following block to the `repositories` section of the `build.gradle` file:

```
repositories {
  maven {
     credentials {
        username "gemfireRepoUsername"
```

```
            password "gemfireRepoPassword"
        }
        url = uri("https://packages.broadcom.com/artifactory/gemfire/")
    }
}
```

2. Add your credentials to the local `.gradle/gradle.properties` or project `gradle.properties` file.

```
gemfireRepoUsername=MY-USERNAME
gemfireRepoPassword=MY-PASSWORD
```

Where:

- `EXAMPLE-USERNAME` is your support.broadcom.com username.
- `MY-PASSWORD` is the Access Token you copied in step 3 in Prerequisites.

Add the Libraries to your Project

After you have set up the repository and credentials, add the Spring Boot for Tanzu GemFire library to your application. To allow for more flexibility with multiple GemFire versions, the Spring Boot for Tanzu GemFire library requires users to add an explicit dependency on the desired version of GemFire.

In the following examples:

- Update the `springBootForGemFire.version` with the version of the library that your project requires.
- Update the `vmwareGemFire.version` with the version of GemFire that your project requires.
- Replace the `spring-boot-3.3-gemfire-10.1` with the version of Spring Boot + the version of GemFire your application requires.

Add the following to your `build.gradle` file.

```
        ext {
            springBootForGemFireVersion = '2.0.3'
            vmwareGemFireVersion = '10.1.2'
        }

        dependencies {
            implementation "com.vmware.gemfire:spring-boot-3.3-gemfire-10.1:$springBoo
tForGemFireVersion"
            implementation "com.vmware.gemfire:gemfire-core:$vmwareGemFireVersion"
            // if using continuous queries
            implementation "com.vmware.gemfire:gemfire-cq:$vmwareGemFireVersion"
        }
```

Spring Enterprise Support

If you have Spring Enterprise support, you are able to select the latest Spring Enterprise versions for your Spring Boot applications. To configure your Spring Boot for Tanzu GemFire application with the latest enterprise versions, follow the steps below.

Configuration

1. Log in to the Broadcom Customer Support Portal with your customer credentials. For more information about login requirements, see the Download Broadcom products and software article.
2. Go to the Spring Enterprise Subscription.
3. Click on “Token Access”.
4. Find the release named **Click Green Token for Repository Access** and click the **Token Download** icon on the right. This opens the instructions on how to use the GemFire artifact repository. At the top, the Access Token is provided. Click **Copy to Clipboard**. You will use this Access Token as the password.

As per the steps described in the GemFire repository setup above, follow the steps to configure your Spring Enterprise repository and credentials for your Maven or Gradle builds.

Setting Spring Enterprise version

Once you have completed the steps to retrieve your repository token and have successfully setup your repository, you can change the Spring project’s version using the following properties.

- spring-boot.version
- spring-data-bom.version
- spring-framework.version
- spring-security.version
- spring-session.version

Maven

```
<properties>
   <spring-boot.version>3.1.14</spring-boot.version>
   <spring-data-bom.version>3.1.13</spring-data-bom.version>
   <spring-framework.version>6.0.26</spring-framework.version>
   <spring-security.version>6.1.12</spring-security.version>
   <spring-session.version>3.1.7</spring-session.version>
</properties>
```

Gradle

```
project.ext.set("spring-boot.version","3.1.14")
project.ext.set("spring-data-bom.version","3.1.13")
project.ext.set("spring-framework.version","6.0.26")
project.ext.set("spring-security.version","6.1.12")
project.ext.set("spring-session.version","3.1.7")
```

Modules

Your application may require more than one module if, for example, you need (HTTP) Session state management, or you need to enable Spring Boot Actuator endpoints for GemFire. You can declare and use any one of the Spring Boot for Tanzu GemFire modules (in addition to the Spring Boot for Tanzu GemFire

library). For a full list of modules and compatible Spring and Tanzu GemFire versions, see Compatibility and Versions.

Building ClientCache Applications

This topic provides an opinionated option for using Spring Boot for VMware Tanzu GemFire.

This opinionated option provided by Spring Boot for Tanzu GemFire is a `ClientCache` instance (see VMware GemFire Java API Reference) created by declaring Spring Boot for Tanzu GemFire on your application classpath.

This topic assumes that most application developers who use Spring Boot to build applications backed by VMware GemFire are building cache client applications deployed in a VMware GemFire Client/Server Topology. The client/server topology is the most common and traditional architecture employed by enterprise applications that use VMware GemFire.

For example, you can begin building a Spring Boot for Tanzu GemFire `ClientCache` application by declaring the Spring Boot for Tanzu GemFire dependency on your application's classpath:

Example 1. Spring Boot for Tanzu GemFire on the application classpath

```
<dependency>
   <groupId>com.vmware.gemfire</groupId>
   <artifactId>spring-boot-3.3-gemfire-10.1</artifactId>
   <version>2.0.0</version>
</dependency>
```

Then you configure and bootstrap your Spring Boot, VMware GemFire `ClientCache` application with the following main application class:

Example 2. Spring Boot, VMware GemFire `ClientCache` Application

```
@SpringBootApplication
public class SpringBootGemFireClientCacheApplication {

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

Your application now has a `ClientCache` instance that can connect to an VMware GemFire server running on `localhost` and listening on the default `CacheServer` port, `40404`.

By default, a VMware GemFire server must be running for the application to use the `ClientCache` instance. However, it is perfectly valid to create a `ClientCache` instance and perform data access operations by using `LOCAL` Regions. This is useful during development.

To develop with `LOCAL` Regions, configure your cache Regions with the `ClientRegionShortcut.LOCAL`data management policy.

When you are ready to switch from your local development environment (IDE) to a client/server architecture in a managed environment, change the data management policy of the client Region from `LOCAL` back to the default (`PROXY`) or even a `CACHING_PROXY`, which causes the data to be sent to and received from one or more servers.

It is uncommon to ever need a direct reference to the `ClientCache` instance provided by Spring Boot for Tanzu GemFire injected into your application components (for example, `@Service` or `@Repository` beans defined in a Spring `ApplicationContext`), whether you are configuring additional VMware GemFire objects (Regions, Indexes, and so on) or are using those objects indirectly in your applications. However, it is 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:

Example 3. Injecting a `GemFireCache` reference

```
@SpringBootApplication
public class SpringBootGemFireClientCacheApplication {

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

    @Bean
    ApplicationRunner runAdditionalClientCacheInitialization(GemFireCache gemfireCach
e) {

        return args -> {

            ClientCache clientCache = (ClientCache) gemfireCache;

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

Auto-configuration

This topic discusses Spring Boot for VMware Tanzu GemFire auto-configuration.

The following Spring Framework, Spring Data for VMware GemFire and Spring Session for VMware GemFire annotations are implicitly declared by Spring Boot for Tanzu GemFire's auto-configuration.

- `@ClientCacheApplication`
- `@EnableGemfireCaching` (alternatively, Spring Framework's `@EnableCaching`)
- `@EnableContinuousQueries`

- `@EnableGemfireFunctionExecutions`
- `@EnableGemfireRepositories`
- `@EnableLogging`
- `@EnablePdx`
- `@EnableSecurity`
- `@EnableSsl`
- `@EnableGemFireHttpSession`

You do not need to explicitly declare any of these annotations on your `@SpringBootApplication` class because they are provided by Spring Boot for Tanzu GemFire. The only reason you would explicitly declare any of these annotations is to override the Spring Boot, and in particular, Spring Boot for Tanzu GemFire's auto-configuration. Otherwise, doing so is unnecessary.

You should read the chapter in Spring Boot's reference documentation on auto-configuration.

Customizing Auto-configuration

You might ask, "How do I customize the auto-configuration provided by Spring Boot for Tanzu GemFire if I do not explicitly declare the annotation?"

For example, you may want to customize the member's name. The `@ClientCacheApplication` annotation provides the `name` attribute so that you can set the client's name. However, Spring Boot for Tanzu GemFire has already implicitly declared the `@ClientCacheApplication` annotation through auto-configuration on your behalf. What do you do?

In this case, Spring Boot for Tanzu GemFire supplies a few additional annotations.

For example, to set the member's name, you can use the `@UseMemberName` annotation:

Example 1. Setting the member's name using `@UseMemberName`

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

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

Example 2. Setting the member's name using the `spring.application.name` property

```
# Spring Boot application.properties

spring.application.name = MyMemberName
```

Example 3. Setting the member's name using the `spring.data.gemfire.cache.name` property

```
# Spring Boot application.properties

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

Note: The spring.data.gemfire.cache.name property is an alias for the `spring.data.gemfire.name` property. Both properties set the name of the client.

In general, there are three ways to customize configuration, even in the context of Spring Boot for Tanzu GemFire's auto-configuration:

- Using annotations provided by Spring Boot for Tanzu GemFire for common and popular concerns (such as naming clients with the `@UseMemberName` annotation or enabling durable clients with the `@EnableDurableClient` annotation).
- Using properties (such as `spring.application.name`, or `spring.data.gemfire.name`, or `spring.data.gemfire.cache.name`).
- Using `configurers` (such as `ClientCacheConfigurer`).

Deactivating Auto-configuration

Spring Boot's reference documentation explains how to deactivate Spring Boot auto-configuration.

Deactivating Auto-configuration also explains how to deactivate Spring Boot for Tanzu GemFire auto-configuration.

In a nutshell, if you want to deactivate any auto-configuration provided by either Spring Boot or Spring Boot for Tanzu GemFire, declare your intent in the `@SpringBootApplication` annotation:

Example 4. Deactivating Specific Auto-configuration Classes

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

Caution: Make sure you understand what you are doing when you deactivate auto-configuration.

Overriding Auto-configuration

Overriding explains how to override Spring Boot for Tanzu GemFire auto-configuration.

In a nutshell, if you want to override the default auto-configuration provided by Spring Boot for Tanzu GemFire, you must annotate your `@SpringBootApplication` class with your intent.

Example 5. Overriding by explicitly declaring `@ClientCacheApplication`

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

You are overriding Spring Boot for Tanzu GemFire's auto-configuration of the `ClientCache` instance. As a result, you have now also implicitly consented to being responsible for other aspects of the configuration (such as security).

Why does that happen?

It happens because, in certain cases, such as security, certain aspects of security configuration (such as SSL) must be configured before the cache instance is created. Also, Spring Boot always applies user configuration before auto-configuration partially to determine what needs to be auto-configured in the first place.

Caution: Make sure you understand what you are doing when you override auto-configuration.

Replacing Auto-configuration

See the Spring Boot reference documentation on replacing auto-configuration.

Understanding Auto-configuration

This section covers the Spring Boot for Tanzu GemFire provided auto-configuration classes that correspond to the Spring Data for VMware GemFire annotations in more detail.

To review the complete list of Spring Boot for Tanzu GemFire auto-configuration classes, see Complete Set of Auto-configuration Classes.

`@ClientCacheApplication`

The Spring Boot for Tanzu GemFire `ClientCacheAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@ClientCacheApplication` annotation.

Spring Boot for Tanzu GemFire starts with the opinion that application developers primarily build GemFire client applications when using Spring Boot. This means building Spring Boot applications with an VMware GemFire `ClientCache` instance connected to a dedicated cluster of VMware GemFire servers that manage the data as part of a client/server topology.

Users do not need to explicitly declare and annotate their `@SpringBootApplication` class with Spring Data for VMware GemFire's `@ClientCacheApplication` annotation. Spring Boot for Tanzu GemFire's provided auto-configuration class is already meta-annotated with Spring Data for VMware GemFire's `@ClientCacheApplication` annotation. Therefore, you only need to do the following:

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

`@EnableGemfireCaching`

The Spring Boot for Tanzu GemFire `CachingProviderAutoConfiguration`class corresponds to the Spring Data for VMware GemFire `@EnableGemfireCaching` annotation.

When using Spring Boot for Tanzu GemFire you don't need to do anything to enable GemFire caching. If you were previously using Spring Data for GemFire, you may remove the `@EnableGemfireCaching` and if you were using plain Spring, you may remove your `GemfireCacheManager` bean.

If you used the core Spring Framework to configure VMware GemFire as a caching provider in Spring's Cache Abstraction, you need to:

Example 7. Configuring caching using the Spring Framework

```
@SpringBootApplication
@EnableCaching
class CachingUsingApacheGemFireConfiguration {

    @Bean
    GemfireCacheManager cacheManager(GemFireCache cache) {

        GemfireCacheManager cacheManager = new GemfireCacheManager();

        cacheManager.setCache(cache);

        return cacheManager;
    }
}
```

If you use Spring Data for VMware GemFire's `@EnableGemfireCaching` annotation, you can simplify the preceding configuration:

Example 8. Configuring caching using Spring Data for VMware GemFire

```
@SpringBootApplication
@EnableGemfireCaching
class CachingUsingApacheGemFireConfiguration {

}
```

Also, if you use Spring Boot for Tanzu GemFire, you need only do:

Example 9. Configuring caching using Spring Boot for Tanzu GemFire

```
@SpringBootApplication
class CachingUsingApacheGemFireConfiguration {

}
```

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

Example 10. Using caching in your application

```
@Service
class CustomerService {

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

See Caching with VMware GemFire for more details.

`@EnableContinuousQueries`

The Spring Boot for Tanzu GemFire `ContinuousQueryAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@EnableContinuousQueries` annotation.

Without having to enable anything, you can annotate your application (POJO) component method(s) with the Spring Data for VMware GemFire `@ContinuousQuery` annotation to register a CQ and start receiving events. The method acts as a `CqEvent` handler or, in VMware GemFire's terminology, the method is an implementation of the `CqListener` interface (see VMware GemFire Java API Reference).

Example 11. Declare application CQs

```
@Component
class MyCustomerApplicationContinuousQueries {

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

As the preceding example shows, you can define the events you are interested in receiving by using an OQL query with a finely tuned query predicate that describes the events of interest and implements the handler method to process the events (such as applying a credit to the customer's account and following up in email).

See Continuous Query for more details.

`@EnableGemfireFunctionExecutions`

The Spring Boot for Tanzu GemFire `FunctionExecutionAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@EnableGemfireFunctionExecutions` annotation.

To execute a `Function`, you need to define the Function execution or implementation in a package below the main `@SpringBootApplication` class:

Example 12. Declare a Function Execution

```
package example.app.functions;

@OnRegion(region = "Accounts")
interface MyCustomerApplicationFunctions {

    void applyCredit(Customer customer);

}
```

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

Example 13. Use the Function

```
package example.app.service;

@Service
class CustomerService {

    @Autowired
    private MyCustomerApplicationFunctions customerFunctions;
```

```
    void analyzeCustomerSentiment(Customer customer) {

        // ...

        this.customerFunctions.applyCredit(customer);

        // ...
    }
}
```

This allows you to focus on defining the logic required by your application and not worry about how Functions are called. Spring Boot for Tanzu GemFire handles this concern for you.

See Function Implementations and Executions for more details.

`@EnableGemfireRepositories`

The Spring Boot for Tanzu GemFire `GemFireRepositoriesAutoConfigurationRegistrar` class corresponds to the Spring Data for VMware GemFire `@EnableGemfireRepositories` annotation.

As with Functions, you only need to focus on the data access operations (such as basic CRUD and simple queries) required by your application to carry out its operation, not with how to create and perform them (for example, `Region.get(key)` and `Region.put(key, obj)`) or execute them (for example, `Query.execute(arguments)`).

Start by defining your Spring Data Repository:

Example 14. Define an application-specific Repository

```
package example.app.repo;

interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findBySentimentEqualTo(Sentiment sentiment);

}
```

Then you can inject the Repository into an application component and use it:

Example 15. 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 focusing only on the data access operations and queries required to carry out the operations of your application.

See Spring Data Repositories for more details.

`@EnableLogging`

The Spring Boot for Tanzu GemFire `LoggingAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@EnableLogging` annotation.

Logging is an essential application concern to understand what is happening in the system along with when and where the events occurred. By default, Spring Boot for Tanzu GemFire auto-configures logging for VMware GemFire with the default log-level, “config”.

You can change any aspect of logging, such as the log-level, in Spring Boot `application.properties`:

Example 16. Change the log-level for VMware GemFire

```
# Spring Boot application.properites.

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

Note: The `spring.data.gemfire.logging.level` property is an alias for `spring.data.gemfire.cache.log-level`.

You can also configure other aspects, such as the log file size and disk space limits for the filesystem location used to store the VMware GemFire log files at runtime.

Under the hood, VMware GemFire’s logging is based on Log4j. Therefore, you can configure VMware GemFire logging to use any logging provider (such as Logback) and configuration metadata appropriate for that logging provider so long as you supply the necessary adapter between Log4j and whatever logging system you use. For instance, if you include `org.springframework.boot:spring-boot-starter-logging`, you are using Logback and you will need the `org.apache.logging.log4j:log4j-to-slf4j` adapter.

`@EnablePdx`

The Spring Boot for Tanzu GemFire `PdxSerializationAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@EnablePdx` annotation.

Any time you need to send an object over the network or overflow or persist an object to disk, your application domain model object must be serializable. It would be painful to have to implement `java.io.Serializable` in every one of your application domain model objects (such as `Customer`) that would potentially need to be serialized.

Furthermore, using Java Serialization may not be ideal (it may not be the most portable or efficient solution) in all cases or even possible in other cases (such as when you use a third-party library over which you have no control).

In these situations, you need to be able to send your object anywhere, anytime without unduly requiring the class type to be serializable and exist on the classpath in every place it is sent. The final destination may

not even be a Java application. This is where VMware GemFire PDX Serialization steps in to help.

However, you do not need to know how to configure PDX to identify the application class types that need to be serialized. Instead, you can define your class type as follows:

Example 17. Customer class

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

    @Id
    private Long id;

    @Indexed
    private String name;

    // ...
}
```

Spring Boot for Tanzu GemFire's auto-configuration handles the rest.

See Data Serialization with PDX for more details.

`@EnableSecurity`

The Spring Boot for Tanzu GemFire `ClientSecurityAutoConfiguration` class corresponds to the Spring Data for VMware GemFire `@EnableSecurity` annotation and applies security (specifically, authentication and authorization (auth) configuration) to clients.

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

Example 18. Supplying Authentication Credentials

```
# Spring Boot application.properties

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

Note: Configuring authentication in a managed environment like Tanzu Platform for Cloud Foundry with GemFire requires no additional setup.

See Security for more details.

`@EnableSsl`

The Spring Boot for Tanzu GemFire `SslAutoConfiguration` class corresponds to the Spring Data for GemFire `@EnableSsl` annotation.

Configuring SSL for secure transport (TLS) between your Spring Boot, VMware GemFire `ClientCache` application and an VMware GemFire cluster can be a real problem, especially to get right from the start. So, it is something that Spring Boot for Tanzu GemFire makes as simple as possible.

You can supply a `trusted.keystore` file containing the certificates in a well-known location (such as the root of your application classpath), and Spring Boot for Tanzu GemFire's auto-configuration steps in to handle the rest.

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

`@EnableGemFireHttpSession`

The Spring Boot for Tanzu GemFire `SpringSessionAutoConfiguration` class corresponds to the Spring Session for VMware GemFire `@EnableGemFireHttpSession` annotation.

Configuring VMware GemFire to serve as the (HTTP) session state caching provider by using Spring Session requires that you only include the correct module, that is `spring-boot-session-3.3-gemfire-10.1:2.0.0`:

Example 19. Using Spring Session

```
<dependency>
    <groupId>com.vmware.gemfire</groupId>
    <artifactId>spring-boot-session-3.3-gemfire-10.1</artifactId>
    <version>2.0.0</version>
</dependency>
```

With Spring Session and specifically Spring Boot for Tanzu GemFire on the classpath of your Spring Boot, VMware GemFire `ClientCache` Web application, you can manage your (HTTP) session state with VMware GemFire. No further configuration is needed. Spring Boot for Tanzu GemFire auto-configuration detects Spring Session on the application classpath and does the rest.

See Spring Session for more details.

RegionTemplateAutoConfiguration

The Spring Boot for Tanzu GemFire `RegionTemplateAutoConfiguration` class has no corresponding Spring Data for VMware GemFire annotation. However, the auto-configuration of a `GemfireTemplate` for every VMware GemFire `Region` defined and declared in your Spring Boot application is still supplied by Spring Boot for Tanzu GemFire.

For example, you can define a Region by using:

Example 20. Region definition using JavaConfig

```
@Configuration
class GemFireConfiguration {

    @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 can define the `Customers` Region by using `@EnableEntityDefinedRegions`:

Example 21. Region definition using `@EnableEntityDefinedRegions`

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

}
```

Then Spring Boot for Tanzu GemFire supplies a `GemfireTemplate` instance that you can use to perform low-level data-access operations (indirectly) on the `Customers` Region:

Example 22. 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 need not explicitly configure `GemfireTemplates` for each Region to which you need low-level data access (such as when you are not using the Spring Data Repository abstraction).

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

See Data Access with GemfireTemplate for more details.

Declarative Configuration

This topic discusses declarative configuration in Spring Boot for VMware Tanzu GemFire.

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 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 that you can 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. It enables features, functionality, services and supporting infrastructure for Spring applications in a loosely integrated way by using conventions (such as the classpath) that ultimately help 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, you can include the `org.springframework.boot:spring-boot-starter-web` dependency on your application classpath. Not only does Spring Boot enable you to build Spring Web MVC Controllers appropriate to your application UC (your responsibility), but it also bootstraps your web application in an embedded Servlet container on startup (Spring Boot's responsibility).

This saves you from having to handle many low-level, repetitive, and tedious development tasks that are error-prone and easy to get wrong when you are trying to solve problems. You need not care how the plumbing works until you need to customize something. And, when you do, you are better informed and prepared to do so.

It is also equally essential that frameworks, such as Spring Boot, get out of the way quickly when application requirements diverge from the provided defaults. This 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 need to use declarative configuration in some cases, whether expressed as bean definitions, in properties, or by some other means. This is so that frameworks do not leave things to chance, especially when things are ambiguous. The framework gives you choice.

Keeping our goals in mind, this chapter:

- Refers you to the Spring Data for VMware GemFire annotations covered by Spring Boot for Tanzu GemFire's auto-configuration.
- Lists all Spring Data for VMware GemFire annotations not covered by Spring Boot for Tanzu GemFire's auto-configuration.

- Covers the Spring Boot for Tanzu GemFire, Spring Session for VMware GemFire and Spring Data for VMware GemFire annotations that you must explicitly declare and that provide the most value and productivity when getting started with VMware GemFire in Spring [Boot] applications.

The list of Spring Data for VMware GemFire annotations covered by Spring Boot for Tanzu GemFire's auto-configuration is discussed in detail in the Appendix: Auto-configuration versus Annotation-based configuration.

To be absolutely clear about which Spring Data for VMware GemFire annotations we are referring to, we mean the Spring Data for VMware GemFire annotations in the `org.springframework.data.gemfire.config.annotation` package.

In subsequent sections, we also cover which annotations are added by Spring Boot for Tanzu GemFire.

Auto-configuration

We explained auto-configuration in detail in the Auto-configuration chapter.

Annotations Not Covered by Auto-configuration

The following Spring Data for VMware GemFire annotations are not implicitly applied by Spring Boot for Tanzu GemFire's auto-configuration:

- `@EnableAutoRegionLookup`
- `@EnableBeanFactoryLocator`
- `@EnableCachingDefinedRegions`
- `@EnableClusterDefinedRegions`
- `@EnableCompression`
- `@EnableDiskStore(s)`
- `@EnableEntityDefinedRegions`
- `@EnableEviction`
- `@EnableExpiration`
- `@EnableGemFireAsLastResource`
- `@EnableGemFireMockObjects`
- `@EnablePool(s)`
- `@EnableStatistics`
- `@UseGemFireProperties`

Productivity Annotations

This section calls out the annotations we believe to be most beneficial for your application development purposes when using VMware GemFire in Spring [Boot] applications.

`@EnableClusterAware`

The `@EnableClusterAware` annotation helps connect to a cluster when running in cloud environments.

Example 1. Declaring `@EnableClusterAware`

```
@SpringBootApplication
@EnableClusterAware
class SpringBootGemFireClientCacheApplication {  }
```

When you annotate your main `@SpringBootApplication` class with `@EnableClusterAware`, your Spring Boot, VMware GemFire `ClientCache` application is able to seamlessly switch between environments with no code or configuration changes, regardless of the runtime environment (such as standalone versus cloud-managed environments).

When a cluster of VMware GemFire servers is detected, the client application sends and receives data to and from the VMware GemFire cluster.

By default, the configuration metadata is sent to the cluster by using a non-secure HTTP connection. However, you can configure HTTPS, change the host and port, and configure the data management policy used by the servers when creating Regions.

`@EnableCachingDefinedRegions`, `@EnableClusterDefinedRegions` and `@EnableEntityDefinedRegions`

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

`@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 are identified by name in the caching annotations are used to create Regions that hold the data you want cached.

Consider the following example:

Example 2. Defining Regions based on Spring or JSR-107 JCache Annotations

```
@Service
class CustomerService {

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

Further consider the following example, in which the main `@SpringBootApplication` class is annotated with `@EnableCachingDefinedRegions`:

Example 3. Using `@EnableCachingDefinedRegions`

```
@SpringBootApplication
@EnableCachingDefinedRegions
class SpringBootGemFireClientCacheApplication {  }
```

With this setup, Spring Boot for Tanzu GemFire would create a client `PROXY` Region named “CustomersByAccountNumber”, as though you created the Region by using either the Java configuration or XML approaches shown earlier.

You can use the `clientRegionShortcut` attribute to change the data management policy of the Regions created on the client.

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

`@EnableEntityDefinedRegions`

As with `@EnableCachingDefinedRegions`, `@EnableEntityDefinedRegions` lets you create Regions based on the entity classes you have defined in your application domain model.

For instance, consider an entity class annotated with Spring Data for VMware GemFire’s `@Region` mapping annotation:

Example 4. Customer entity class annotated with `@Region`

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

    @Id
    private Long id;

    @Indexed
    private String name;

}
```

For this class, Spring Boot for Tanzu GemFire creates Regions from the name specified in the `@Region` mapping annotation on the entity class. In this case, the `Customer` application-defined entity class results in the creation of a Region named “Customers” when the main `@SpringBootApplication` class is annotated with `@EnableEntityDefinedRegions`:

Example 5. Using `@EnableEntityDefinedRegions`

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

As with the `@EnableCachingDefinedRegions` annotation, you can set the client Region data management policy by using the `clientRegionShortcut` attribute, and set a dedicated `Pool` of connections used by client Regions with the `poolName` attribute.

However, unlike the `@EnableCachingDefinedRegions` annotation, you must specify either the `basePackage` attribute or the type-safe `basePackageClasses` attribute (recommended) when you use 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, such as Hibernate.

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

Both the `basePackages` and `basePackageClasses` attributes accept an array of values. With `basePackageClasses`, you need only refer to a single class type in that package and every class in that package as well as classes in the sub-packages are 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 to be an entity.

For example, suppose you had the following structure:

Example 6. 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:

Example 7. Targeting with `@EnableEntityDefinedRegions`

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

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

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

`@EnableClusterDefinedRegions`

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

To do so, annotate your main `@SpringBootApplication` class with `@EnableClusterDefinedRegions`:

Example 8. Using `@EnableClusterDefinedRegions`

```
@SpringBootApplication
@EnableClusterDefinedRegions
```

```
class SpringBootGemFireClientCacheApplication {  }
```

Every Region that exists on the servers in the VMware GemFire cluster 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 a client `PROXY` Region with the same name (“ServerRegion”) into your Spring Boot application:

Example 9. Using a server-side Region on the client

```
@Component
class SomeApplicationComponent {

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

    public void someMethod() {

        EntityType entity = new EntityTypeImpl(...);

        this.serverRegion.put(1, entity);

        // ...
    }
}
```

Spring Boot for Tanzu GemFire auto-configures a `GemfireTemplate` for the “ServerRegion” Region (see RegionTemplateAutoConfiguration, so a better way to interact with the client `PROXY` Region that corresponds to the “ServerRegion” Region on the server is to inject the template:

Example 10. Using a server-side Region on the client with a template

```
@Component
class SomeApplicationComponent {

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

    public void someMethod() {

        EntityType entity = new EntityTypeImpl(...);

        this.serverRegionTemplate.put(1, entity);

        //...
    }
}
```

`@EnableExpiration`

It is often useful to define both eviction and expiration policies, particularly with a system like VMware GemFire, because it primarily keeps data in memory (on the JVM Heap). Your data volume size may far exceed the amount of available JVM Heap memory, and keeping too much data on the JVM Heap can cause Garbage Collection (GC) issues.

Defining eviction and expiration policies lets you limit what is kept in memory and for how long.

With Spring Data for VMware GemFire, you can define the expiration policies associated with a particular application class type on the class type itself, by using the `@Expiration`, `@IdleTimeoutExpiration` and `@TimeToLiveExpiration` annotations.

See the VMware GemFire User Guide for more details on the different expiration types — that is *Idle Timeout* (TTI) versus *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 (for example, the last time a `Customer` was read). To do so, we can define an idle timeout expiration (TTI) policy on our `Customer` class type:

Example 11. Customer entity class with Idle Timeout Expiration (TTI)

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

    @Id
    private Long id;

    @Indexed
    private String name;

}
```

The `Customer` entry in the `Customers` Region is `invalidated` after 300 seconds (5 minutes).

To enable annotation-based expiration policies, we need to annotate our main `@SpringBootApplication` class with `@EnableExpiration`:

Example 12. Enabling Expiration

```
@SpringBootApplication
@EnableExpiration
class SpringBootGemFireApplication {  }
```

Note: Technically, this entity-class-specific annotation-based expiration policy is implemented by using VMware GemFire's `CustomExpiry` interface. For more information, see the VMware GemFire Java API Reference.

Externalized Configuration

This topic discusses externalized configuration of Spring Boot for VMware Tanzu GemFire.

Like Spring Boot, Spring Boot for Tanzu GemFire supports externalized configuration. For more information about Spring Boot, see the Spring Boot documentation.

Externalized configuration is configuration metadata stored in Spring Boot `application.properties`. You can separate concerns by addressing each concern in an individual properties file. Optionally, you can enable any specific property file for only a specific profile. For more information about Spring Boot profiles, see profile in the Spring Boot documentation.

You can do many other powerful things, such as using placeholders in properties, encrypting properties, and so on.

This topic focuses particularly on type safety.

Like Spring Boot, Spring Boot for Tanzu GemFire provides a hierarchy of classes that captures configuration for several VMware GemFire features in an associated `@ConfigurationProperties` annotated class. Again, the configuration metadata is specified as well-known, documented properties in one or more Spring Boot `application.properties` files.

For instance, a Spring Boot, VMware GemFire `ClientCache` application might be configured as follows:

Example 1. Spring Boot `application.properties` containing Spring Data properties for VMware GemFire

```
# Spring Boot application.properties used to configure VMware GemFire

spring.data.gemfire.name=MySpringBootGemFireApplication

# 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]
```

You may sometimes 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. You can access any property by using Spring's `Environment` abstraction:

Example 2. Using the Spring `Environment`

```
@Configuration
class GemFireConfiguration {

    void readConfigurationFromEnvironment(Environment environment) {
        boolean copyOnRead = environment.getProperty("spring.data.gemfire.cache.copy-o
n-read",
            Boolean.TYPE, false);
    }
}
```

While using `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, you can now, thanks to Spring Boot for Tanzu GemFire's auto-configured configuration processor, access the configuration metadata by using `@ConfigurationProperties` classes.

To add to the preceding example, you can now do the following:

Example 3. 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 that are used to configure VMware GemFire in a Spring context. You need only autowire an instance of `GemFireProperties` into your application component.

Externalized Configuration of Spring Session

You can access the externalized configuration of Spring Session when you use VMware GemFire as your (HTTP) session state caching provider.

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

As shown earlier in this chapter, you can specify Spring Session for VMware GemFire properties as follows:

Example 4. Spring Boot `application.properties` for Spring Session using VMware GemFire as the (HTTP) session state caching provider

```
# Spring Boot application.properties used to configure VMware GemFire as a (HTTP) sess
ion 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, you can do something similar to the following example:

Example 5. Using `SpringSessionProperties`

```
@Component
class MyApplicationComponent {

    @Autowired
    private SpringSessionProperties springSessionProperties;

    public void someMethodUsingSpringSessionProperties() {

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

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

Using GemFire Properties

This topic discusses using GemFire properties with Spring Boot for Tanzu GemFire.

You can declare VMware GemFire properties from `gemfire.properties` in Spring Boot `application.properties`.

Tip	See the User Guide for a complete list of valid VMware GemFire properties.

Note that you can declare only valid GemFire properties in `gemfire.properties` or, alternatively, `gfsecurity.properties`.

The following example shows how to declare properties in `gemfire.properties`:

Example 1. Valid `gemfire.properties`

```
# GemFire Properties in gemfire.properties

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

All the properties declared in the preceding example correspond to valid GemFire properties. It is illegal to declare properties in `gemfire.properties` that are not valid GemFire properties, even if those properties are prefixed with a different qualifier (such as `spring.*`). VMware GemFire throws an `IllegalArgumentException` for invalid properties.

Consider the following `gemfire.properties` file with an `invalid-property`:

Example 2. Invalid `gemfire.properties`

```
# GemFire Properties in gemfire.properties

name=ExampleCacheName
invalid-property=TEST
```

VMware GemFire throws an `IllegalArgumentException`:

Example 3. `IllegalArgumentException` thrown by VMware GemFire for Invalid Property (Full Text Omitted)

```
Exception in thread "main" java.lang.IllegalArgumentException: Unknown configuration a
ttribute name invalid-property.
Valid attribute names are: ack-severe-alert-threshold ack-wait-threshold archive-disk-
space-limit ...
    at o.a.g.internal.AbstractConfig.checkAttributeName(AbstractConfig.java:333)
```

```
    at o.a.g.distributed.internal.AbstractDistributionConfig.checkAttributeName(Abstra
ctDistributionConfig.java:725)
    at o.a.g.distributed.internal.AbstractDistributionConfig.getAttributeType(Abstract
DistributionConfig.java:887)
    at o.a.g.internal.AbstractConfig.setAttribute(AbstractConfig.java:222)
    at o.a.g.distributed.internal.DistributionConfigImpl.initialize(DistributionConfig
Impl.java:1632)
    at o.a.g.distributed.internal.DistributionConfigImpl.<init>(DistributionConfigImp
l.java:994)
    at o.a.g.distributed.internal.DistributionConfigImpl.<init>(DistributionConfigImp
l.java:903)
    at o.a.g.distributed.internal.ConnectionConfigImpl.lambda$new$2(ConnectionConfigIm
pl.java:37)
    at o.a.g.distributed.internal.ConnectionConfigImpl.convert(ConnectionConfigImpl.ja
va:73)
    at o.a.g.distributed.internal.ConnectionConfigImpl.<init>(ConnectionConfigImpl.jav
a:36)
    at o.a.g.distributed.internal.InternalDistributedSystem$Builder.build(InternalDist
ributedSystem.java:3004)
    at o.a.g.distributed.internal.InternalDistributedSystem.connectInternal(InternalDi
stributedSystem.java:269)
    at o.a.g.cache.client.ClientCacheFactory.connectInternalDistributedSystem(ClientCa
cheFactory.java:280)
    at o.a.g.cache.client.ClientCacheFactory.basicCreate(ClientCacheFactory.java:250)
    at o.a.g.cache.client.ClientCacheFactory.create(ClientCacheFactory.java:216)
    at org.example.app.ApacheGeodeClientCacheApplication.main(...)
```

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

Additionally, because of VMware GemFire's constraint on properties, you cannot use the full power of Spring Boot when you compose `application.properties`.

You can include certain properties based on a Spring profile while excluding other properties. This is essential when properties are environment- or context-specific.

Spring Data for VMware GemFire already provides a wide range of properties mapping to VMware GemFire properties.

For example, the Spring Data for VMware GemFire `spring.data.gemfire.locators` property maps to the `gemfire.locators` property (`locators` in `gemfire.properties`) from VMware GemFire. Likewise, there are a full set of Spring Data for VMware GemFire properties that map to the corresponding VMware GemFire properties in the Appendix.

You can express the GemFire properties shown earlier as Spring Data for VMware GemFire properties in Spring Boot `application.properties`, as follows:

Example 4. Configuring GemFire Properties using Spring Data for VMware GemFire Properties

```
# Spring Data for VMware GemFire properties in application.properties

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

However, some VMware GemFire properties have no equivalent Spring Data for VMware GemFire property, such as `gemfire.groups` (`groups` in `gemfire.properties`). This is partly because many VMware GemFire properties are applicable only when configured on the server (such as `groups` or `enforce-unique-host`).

Furthermore, many of the Spring Data for VMware GemFire properties also correspond to API calls.

For example, `spring.data.gemfire.cache.client.keep-alive` translates to the `ClientCache.close(boolean keepAlive)` API call (see VMware GemFire Java API Reference).

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

Therefore, you can now declare VMware GemFire properties in Spring Boot `application.properties` directly, as follows:

Example 5. GemFire Properties declared in Spring Boot `application.properties`

```
# Spring Boot application.properties

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

This is convenient and ideal for several reasons:

- If you already have a large number of VMware GemFire properties declared as `gemfire.` properties (either in `gemfire.properties` or `gfsecurity.properties`) or declared on the Java command-line as JVM System properties (such as `-Dgemfire.name=ExampleCacheName`), you can reuse these property declarations.
- If you are unfamiliar with Spring Data for VMware GemFire's corresponding properties, you can declare GemFire properties instead.
- You can take advantage of Spring features, such as Spring profiles.
- You can also use property placeholders with GemFire properties (such as `gemfire.log-level=${external.log-level.property}`).

Tip	We encourage you to use the Spring Data for VMware GemFire properties, which cover more than VMware GemFire properties.

However, Spring Boot for Tanzu GemFire requires that the GemFire property must have the `gemfire.` prefix in Spring Boot `application.properties`. This indicates that the property belongs to VMware GemFire. Without the `gemfire.` prefix, the property is not appropriately applied to the VMware GemFire cache instance.

It would be ambiguous if your Spring Boot applications integrated with several technologies, including VMware GemFire, and they too had matching properties, such as `bind-address` or `log-file`.

Spring Boot for Tanzu GemFire makes a best attempt to log warnings when a GemFire property is invalid or is not set. For example, the following GemFire property would result in logging a warning:

Example 6. Invalid VMware GemFire Property

```
# Spring Boot application.properties

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

The resulting warning in the log would read:

Example 7. Invalid GemFire Property Warning Message

```
[gemfire.non-existing-property] is not a valid VMware GemFire property
```

If a GemFire Property is not properly set, the following warning is logged:

Example 8. Invalid GemFire Property Value Warning Message

```
VMware GemFire Property [gemfire.security-manager] was not set
```

In regard to the third point mentioned earlier, you can now compose and declare GemFire properties based on a context (such as your application environment) using Spring profiles.

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

Example 9. Base Properties

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

Then you can vary the properties by environment, as the next two listings (for QA and production) show:

Example 10. QA Properties

```
# Spring Boot application-qa.properties

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

Example 11. Production Properties

```
# Spring Boot application-prod.properties

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

You can then apply the appropriate set of properties by configuring the Spring profile with `-Dspring.profiles.active=prod`. You can also enable more than one profile at a time with `-`

`Dspring.profiles.active=profile1,profile2,…,profileN`

If both `spring.data.gemfire.*` properties and the matching VMware GemFire properties are declared in Spring Boot `application.properties`, the Spring Data for VMware GemFire properties take precedence.

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

Consider the following Spring Boot `application.properties`:

Example 12. Property Precedence

```
# Spring Boot application.properties

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

The `durable-client-id` is `987`. It does not matter which order the Spring Data for VMware GemFire or VMware GemFire properties are declared in Spring Boot `application.properties`. The matching Spring Data for VMware GemFire property overrides the VMware GemFire property when duplicates are found.

Finally, you cannot refer to GemFire properties declared in Spring Boot `application.properties` with the Spring Boot for Tanzu GemFire `GemFireProperties` class.

Consider the following example:

Example 13. GemFire Properties declared in Spring Boot `application.properties`

```
# Spring Boot application.properties

gemfire.name=TestCacheName
```

Given the preceding property, the following assertion holds:

```
import org.springframework.geode.boot.auto-configure.configuration.GemFireProperties;

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

    @Autowired
    private GemFireProperties gemfireProperties;

    @Test
    public void gemfirePropertiesTestCase() {
        assertThat(this.gemfireProperties.getCache().getName()).isNotEqualTo("TestCach
eName");
    }
}
```

Tip	You can declare `application.properties` in the `@SpringBootTest` annotation. For example, you could have declared `gemfire.name` in the annotation by setting `@SpringBootTest(properties = { "gemfire.name=TestCacheName" })` for testing purposes instead of declaring the property in a separate Spring Boot `application.properties` file.

Only `spring.data.gemfire.*` prefixed properties are mapped to the Spring Boot for Tanzu GemFire `GemFireProperties` class hierarchy.

Caching with VMware GemFire

This topic discusses using VMware GemFire as a cache provider for use with Spring Boot for VMware Tanzu GemFire.

One of the simplest, quickest, and least invasive ways to start using VMware GemFire in your Spring Boot applications is to use VMware GemFire as a caching provider in Spring's Cache Abstraction. Spring Data for VMware GemFire enables VMware GemFire to function as a caching provider in Spring's Cache Abstraction.

Note: VMware highly recommends that you thoroughly understand the concepts behind Spring's Cache Abstraction before you continue.

Caching can be an 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: looking up a customer by name or account number, looking up a book by ISBN, geocoding a physical address, and caching the calculation of a person's credit score when the person applies for a financial loan.

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

Spring's declarative, annotation-based caching also supports JSR-107 JCache annotations.

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

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

Example 1. Spring application service component applicable to caching

```
@Service
class FinancialLoanApplicationService {

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

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

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

Spring Boot for Tanzu GemFire auto-configures VMware GemFire as the caching provider when VMware GemFire is declared on the application classpath and when no other caching provider has been configured.

If Spring Boot for Tanzu GemFire detects that another cache provider has already been configured, then VMware GemFire will not function as the caching provider for the application. This lets you configure another store as the caching provider and perhaps use VMware GemFire as your application's persistent store.

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

To configure the necessary Regions that back the caches declared in Spring's cache annotations, use Spring Data for VMware GemFire's `@EnableCachingDefinedRegions` annotation.

The following listing shows a complete Spring Boot application:

Example 2. Spring Boot cache enabled application using VMware GemFire

```
package example.app;

@SpringBootApplication
@EnableCachingDefinedRegions
class FinancialLoanApplication {

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

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

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

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

Look-Aside Caching, Near Caching, and Inline Caching

Four different types of caching patterns can be applied with Spring when using VMware GemFire for your application caching needs:

- Look-aside caching

- Near caching
- Inline caching

Typically, when most users think of caching, they think 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 (no extra network hops). This also improves application throughput — that is, the amount of work completed in a given period of time.

Within *Inline caching*, developers have a choice between synchronous (read/write-through) and asynchronous (write-behind) configurations depending on the application use case and requirements. Synchronous, read/write-through Inline caching is necessary if consistency is a concern. Asynchronous, write-behind Inline caching is applicable if throughput and low-latency are a priority.

Look-Aside Caching

See the corresponding sample guide and code to see Look-aside caching with VMware GemFire in action.

The caching pattern demonstrated in the preceding example is a form of Look-aside caching (or "*Cache Aside*").

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

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

The typical Look-aside caching pattern applied in your Spring application code looks similar to the following:

Example 3. Look-Aside Caching Pattern Applied

```
@Service
class CustomerService {

  private final CustomerRepository customerRepository;

  @Cacheable("Customers")
  public 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 that accesses the external data source (for example, an RDBMS) directly. The `@Cacheable` annotation wraps, or "decorates", the `findByAccount(:Account):Customer` operation (method) to provide caching behavior.

This operation may be expensive because it may validate the customer's account before looking up the customer, pull multiple bits of information to retrieve the customer record, or perform other actions that require significant resources. This expense is a reason for caching.

Near Caching

To see a Near caching example, see the corresponding sample in Near Caching with Spring and nearin GitHub.

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

We already mentioned that Spring Boot for Tanzu GemFire provides an auto-configured `ClientCache` instance by default. A `ClientCache` instance is most effective when the data access operations, including cache access, are distributed to the servers in a cluster that is accessible to the client and, in most cases, multiple clients. This lets other cache client applications access the same data. However, this also means the application incurs 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 to maintain a subset of the data in the corresponding server-side cache (a Region). Therefore, the client cache contains only the data of interest to the application. This "local" cache (a client-side Region) is consulted before forwarding the lookup request to the server.

To enable Near caching when using VMware GemFire, change the Region's (that is the `Cache` in Spring's Cache Abstraction) data management policy from `PROXY` (the default) to `CACHING_PROXY`:

Example 4. Enable Near Caching with VMware GemFire

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

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

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

See also the VMware GemFire documentation concerning client/server event distribution and, specifically, "Client Interest Registration on the Server," which applies when you use 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 that have access to the same data.

Inline Caching

The next pattern of caching covered in this chapter is Inline caching.

You can apply two different configurations of Inline caching to your Spring Boot applications when you use the Inline caching pattern: synchronous (read/write-through) and asynchronous (write-behind).

Note: Asynchronous offers only write capabilities, from the cache to the external data source. No option exists to asynchronously and automatically load the cache when the value becomes available in the external data source.

Synchronous Inline Caching

See the corresponding sample guide and code to see Inline caching with VMware GemFire in action.

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

With VMware GemFire, you can configure the cache (or, to use VMware GemFire terminology, the Region) with a `CacheLoader` (see VMware GemFire Java API Reference). A `CacheLoader` is implemented to retrieve missing values from an external data source when a cache miss occurs. The external data source could be an RDBMS or any other type of data store.

For more information about data loaders, see How Data Loaders Work in the VMware GemFire User's Guide.

Similarly, you can also configure an VMware GemFire Region with a `CacheWriter` (see VMware GemFire Java API Reference). A `CacheWriter` is responsible for writing an entry that has been 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, the entry is not stored in the Region. This helps to ensure consistency between the backend data store and the VMware GemFire Region.

You can also implement Inline caching using asynchronous write-behind operations by registering an `AsyncEventListener` on an `AsyncEventQueue` attached to a server-side Region. For more details, see Implementing an AsyncEventListener for Write-Behind Cache Event Handling in the VMware GemFire product documentation. 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:

Example 5. Inline Caching Pattern Applied

```
@Service
class CustomerService {

  private CustomerRepository customerRepository;

  Customer findByAccount(Account account) {

      // pre-processing logic here

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

      // post-processing logic here.

      return customer;
  }
}
```

The main difference is that no Spring or JSR-107 caching annotations are applied to the application's service methods, and the `CustomerRepository` accesses VMware GemFire directly and the RDBMS

indirectly.

Implementing CacheLoaders and 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 or writer to a Region. Given that 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 or writer.

```
@SpringBootApplication
class FinancialLoanApplication {

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

    @Bean("EligibilityDecisions")
    ClientRegionFactoryBean<?, ?> eligibilityDecisionsRegion(
            GemFireCache gemfireCache, CacheLoader eligibilityDecisionLoader,
            CacheWriter eligibilityDecisionWriter) {

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

        eligibilityDecisionsRegion.setCache(gemfireCache);
        eligibilityDecisionsRegion.setCacheLoader(eligibilityDecisionLoader);
        eligibilityDecisionsRegion.setCacheWriter(eligibilityDecisionWriter);
        eligibilityDecisionsRegion.setPersistent(false);

        return eligibilityDecisionsRegion;
    }

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

        return new EligibilityDecisionLoader(dataSource);
    }

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

        return new EligibilityDecisionWriter(dataSource);
    }

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

Then you could implement the `CacheLoader` and `CacheWriter` interfaces, as appropriate. For more information about the `CacheLoader` and `CacheWriter` interfaces, see VMware GemFire Java API Reference.

Example 6. EligibilityDecisionLoader

```
class EligibilityDecisionLoader implements CacheLoader<Object, EligibilityDecision> {

  private final DataSource dataSource;

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

  public EligibilityDecision load(LoaderHelper<Object, EligibilityDecision> helper) {

    Object key = helper.getKey();

    // Use the configured DataSource to load the EligibilityDecision identified by the
key
    // from a backend, external data store.
  }
}
```

Spring Boot for Tanzu GemFire provides the `org.springframework.geode.cache.support.CacheLoaderSupport` `@FunctionalInterface` to conveniently implement application `CacheLoaders`.

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

Example 7. EligibilityDecisionWriter

```
class EligibilityDecisionWriter implements CacheWriter<Object, EligibilityDecision> {

  private final DataSource dataSource;

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

   public void beforeCreate(EntryEvent<Object, EligibilityDecision> event) {
    // Use configured DataSource to save (e.g. INSERT) the entry value into the backen
d data store
  }

  public void beforeUpdate(EntryEvent<Object, EligibilityDecision> event) {
    // Use the configured DataSource to save (e.g. UPDATE or UPSERT) the entry value i
nto the backend data store
  }

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

  // ...
}
```

Spring Boot for Tanzu GemFire provides the `org.springframework.geode.cache.support.CacheWriterSupport` interface to conveniently implement

application `CacheWriters`.

Note: Your `CacheWriter` implementation can use any data access technology to interface with your backend data store (for example JDBC, Spring's `JdbcTemplate`, JPA with Hibernate, and others). It is not limited to using only a `javax.sql.DataSource`. This topic presents another more useful and convenient approach to implementing Inline caching in the next section.

Inline Caching with Spring Data Repositories

Spring Boot for Tanzu GemFire offers dedicated support to configure Inline caching with Spring Data Repositories.

This is powerful, because it lets you:

- Access any backend data store supported by Spring Data.
- Use complex mapping strategies (such as ORM provided by JPA with Hibernate).

We believe that users should store data where it is most easily accessible. If you access and process documents, then a document store is probably going to be the most logical choice to manage your application's documents.

However, this does not mean that you have to give up VMware GemFire in your application/system architecture. You can use each data store for what it is good at. While document stores are excellent at handling documents, VMware GemFire is a valuable choice for consistency, high-availability/low-latency, high-throughput, scale-out application use cases.

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

Suppose you use JPA and Hibernate to access data managed in an Oracle database. Then, you can configure VMware GemFire 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:

Example 8. Inline caching configuration using Spring Boot for Tanzu GemFire

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

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

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

Spring Boot for Tanzu GemFire 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` configures and adapts the Spring Data `CrudRepository` as a `CacheLoader` and `CacheWriter` registered on the Region (for example, “Customers”) to enable Inline caching functionality.

You need only 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 and retrieve data stored in the cache (Region) to and from a backend database. However, you can inject any Spring Data Repository for any data store that supports the Spring Data Repository abstraction.

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

About `RepositoryAsyncEventListener`

The Spring Boot for Tanzu GemFire `RepositoryAsyncEventListener` class is the magic ingredient behind the integration of the cache with an external data source.

The listener is a specialized adapter that processes `AsyncEvents` by invoking an appropriate `CrudRepository` method based on the cache operation. The listener requires an instance of `CrudRepository`. The listener supports any external data source supported by Spring Data’s Repository abstraction.

Backend data store, data access operations (such as INSERT, UPDATE, DELETE, and so on) triggered by cache events are performed asynchronously from the cache operation. This means the state of the cache and backend data store will be “eventually consistent”.

Given the complex nature of “eventually consistent” systems and asynchronous concurrent processing, the `RepositoryAsyncEventListener` lets you register a custom `AsyncEventErrorHandler` to handle the errors that occur during processing of `AsyncEvents`, perhaps due to a faulty backend data store data access operation (such as `OptimisticLockingFailureException`), in an application-relevant way.

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

Example 9. AsyncEventErrorHandler interface definition

```
@FunctionalInterface
interface AsyncEventErrorHandler extends Function<AsyncEventError, Boolean> { }
```

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

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

Example 13. Custom `AsyncEventErrorHandler` implementation

```
class CustomAsyncEventErrorHandler implements AsyncEventErrorHandler {

    @Override
    public Boolean apply(AsyncEventError error) {

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

        return false;
    }
}
```

You can configure the `RepositoryAsyncEventListener` with your custom `AsyncEventErrorHandler` by using the `AsyncInlineCachingRegionConfigurer`:

Example 10. Configuring a custom `AsyncEventErrorHandler`

```
@Configuration
class GemFireConfiguration {

    @Bean
    CustomAsyncEventErrorHandler customAsyncEventErrorHandler() {
        return new CustomAsyncEventErrorHandler();
    }

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

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

Also, since `AsyncEventErrorHandler` implements `Function`, you can compose multiple error handlers by using `Function.andThen(:Function)`.

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

`CREATE` and `UPDATE` translate to `CrudRepository.save(entity)`. The `entity` is derived from `AsyncEvent.getDeserializedValue()`.

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

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

This interface becomes useful if and when you want to implement `CrudRepository` method invocations for other `AsyncEvent Operations` not handled by Spring Boot for Tanzu GemFire's `RepositoryAsyncEventListener`.

The `AsyncEventOperationRepositoryFunction` interface is defined as follows:

Example 11. AsyncEventOperationRepositoryFunction interface definition

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

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

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

For convenience, Spring Boot for Tanzu GemFire provides the `AbstractAsyncEventOperationRepositoryFunction` class for extension, where you can provide implementations for the `cacheProcess(:AsyncEvent)` and `doRepositoryOp(entity)` methods.

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

For example, you may want to handle `Operation.INVALIDATE` cache (see VMware GemFire Java API Reference) events as well, deleting the entity from the backend data store by invoking the `CrudRepository.delete(entity)` method:

Example 12. Handling `AsyncEvent`, `Operation.INVALIDATE`

```
@Component
class InvalidateAsyncEventRepositoryFunction
        extends RepositoryAsyncEventListener.AbstractAsyncEventOperationRepositoryFunc
tion<Object, Object> {

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

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

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

You can then register your user-defined, `AsyncEventOperationRepositoryFunction` (that is, `InvalidateAsyncEventRepositoryFunction`) with the `RepositoryAsyncEventListener` by using the `AsyncInlineCachingRegionConfigurer`:

Example 13. Configuring a user-defined `AsyncEventOperationRepositoryFunction`

```
import org.springframework.geode.cache.RepositoryAsyncEventListener;

@Configuration
class GemFireConfiguration {

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

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

                if (listener instanceof RepositoryAsyncEventListener) {

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

                    repositoryListener.register(new InvalidAsyncEventRepositoryFunctio
n(repositoryListener));
                }

                return listener;
            });
    }
}
```

This same technique can be applied to `CREATE`, `UPDATE`, and `REMOVE` cache operations as well, effectively overriding the default behavior for these cache operations handled by Spring Boot for Tanzu GemFire.

About `AsyncInlineCachingRegionConfigurer`

As we saw in the previous section, you can intercept and post-process the essential components that are constructed and configured by the `AsyncInlineCachingRegionConfigurer` class during initialization.

Spring Boot for Tanzu GemFire's lets you intercept and post-process the `AsyncEventListener` (such as `RepositoryAsyncEventListener`), the `AsyncEventQueueFactory` and even the `AsyncEventQueue` created by the `AsyncInlineCachingRegionConfigurer` (a Spring Data for VMware GemFire `RegionConfigurer`) during Spring `ApplicationContext` bean initialization.

The `AsyncInlineCachingRegionConfigurer` class provides the following builder methods to intercept and post-process any of the following VMware GemFire objects:

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

All of these `apply*` methods accept a `java.util.function.Function` that applies the logic of the `Function` to the VMware GemFire object (such as `AsyncEventListener`), returning the object as a result.

Note: The VMware GemFire object returned by the `Function` may be the same object, a proxy, or a completely new object. Essentially, the returned object can be anything you want. This is the fundamental premise behind Aspect-Oriented Programming (AOP) and the Decorator software design pattern.

The `apply*` methods and the supplied `Function` let you decorate, enhance, post-process, or otherwise modify the VMware GemFire objects created by the configurer.

The `AsyncInlineCachingRegionConfigurer` strictly adheres to the open/close principle and is, therefore, flexibly extensible.

Advanced Caching Configuration

VMware GemFire supports additional caching capabilities to manage the entries stored in the cache.

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

You can employ several techniques to more effectively manage memory, such as using eviction, possibly overflowing data to disk, configuring both entry Idle-Timeout_ (TTI) and Time-to-Live_ (TTL) expiration policies, configuring compression, and using off-heap or main memory.

You can use several other strategies as well, as described in Managing Heap and Off-heap Memory.

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 VMware GemFire.

You can specifically call out your Spring Cache Abstraction provider by using the `spring.cache.type` property in `application.properties`:

Example 14. Disable Spring's Cache Abstraction

```
#application.properties

spring.cache.type=none
...
```

For more details, see the Spring Boot documentation.

Note: Spring Boot does not properly recognize `spring.cache.type=[gemfire|geode]`, even though Spring Boot for Tanzu GemFire is set to handle either of the `gemfire` or `geode` property values.

Data Access with GemfireTemplate

This topic explains how to access data stored in GemFire when using Spring Boot for VMware Tanzu GemFire.

There are several ways to access data stored in VMware GemFire.

For example, you can use the `Region API` (see VMware GemFire Java API Reference) directly. If you are driven by the application's domain context, you can use the power of Spring Data Repositories instead.

While the Region API offers flexibility, it couples your application to VMware GemFire. While using Spring Data Repositories provides a very powerful and convenient abstraction, you give up the flexibility provided by a lower-level Region API.

A good compromise is to use the Template software design pattern. This pattern is consistently and widely used throughout the entire Spring portfolio.

For example, the Spring Framework provides `JdbcTemplate` and `JmsTemplate`.

Spring Data for VMware GemFire offers the `GemfireTemplate`.

The `GemfireTemplate` provides a highly consistent and familiar API to perform data access operations on VMware GemFire cache `Regions`.

`GemfireTemplate` offers:

- A simple and convenient data access API to perform basic CRUD and simple query operations on cache Regions.
- Use of Spring Framework's consistent data access Exception hierarchy.
- Automatic enlistment in the presence of local cache transactions.
- Consistency and protection from `Region API` (see VMware GemFire Java API Reference) breaking changes.

Given these advantages, Spring Boot for Tanzu GemFire auto-configures `GemfireTemplate` beans for each Region present in the VMware GemFire cache.

Additionally, Spring Boot for Tanzu GemFire is careful not to create a `GemfireTemplate` if you have already declared a `GemfireTemplate` bean in the Spring `ApplicationContext` for a given Region.

Explicitly Declared Regions

Consider an explicitly declared Region bean definition: 1. Explicitly Declared Region Bean Definition

```
@Configuration
class GemFireConfiguration {

    @Bean("Example")
```

```
    ClientRegionFactoryBean<?, ?> exampleRegion(GemFireCache gemfireCache) {
        // ...
    }
}
```

Spring Boot for Tanzu GemFire automatically creates a `GemfireTemplate` bean for the `Example` Region by using the bean name `exampleTemplate`. Spring Boot for Tanzu GemFire names the `GemfireTemplate` bean after the Region by converting the first letter in the Region's name to lowercase and appending `Template` to the bean name.

In a managed Data Access Object (DAO), you can inject the Template:

```
@Repository
class ExampleDataAccessObject {

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

}
```

You should use the `@Qualifier` annotation to qualify which `GemfireTemplate` bean you are specifically referring, especially if you have more than one Region bean definition.

Entity-defined Regions

Spring Boot for Tanzu GemFire auto-configures `GemfireTemplate` beans for entity-defined Regions.

Consider the following entity class:

Example 1. Customer class

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

Further, consider the following configuration:

Example 2. VMware GemFire Configuration

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

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

Example 3. CustomerService application component

```
@Service
class CustomerService {

    @Autowired
```

```
    @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 you use the `@EnableEntityDefineRegions` annotation.

Caching-defined Regions

Spring Boot for Tanzu GemFire auto-configures `GemfireTemplate` beans for caching-defined Regions.

When you use Spring Framework's Cache Abstraction backed by VMware GemFire, one requirement is to configure Regions for each of the caches specified in the caching annotations of your application service components.

Fortunately, Spring Boot for Tanzu GemFire makes enabling and configuring caching easy and automatic.

Consider the following cacheable application service component:

Example 4. Cacheable `CustomerService` class

```
@Service
class CacheableCustomerService {

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

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

Further, consider the following configuration:

Example 5. VMware GemFire Configuration

```
@Configuration
@EnableCachingDefinedRegions
class GemFireConfiguration {

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

Spring Boot for Tanzu GemFire auto-configures a `GemfireTemplate` bean named `customersByNameTemplate` to perform data access operations on the `CustomersByName` (`@Cacheable`) Region. You can then inject the bean into any managed application component, as shown in the preceding application service component example.

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

Warning: Autowiring, or *injecting* `GemfireTemplate` beans auto-configured by Spring Boot for Tanzu GemFire for caching-defined Regions into your application components does not always work. This has to do with the Spring container bean creation process. In those cases, you may need to look up the `GemfireTemplate` by using `applicationContext.getBean("customersByNameTemplate", GemfireTemplate.class)`. This works when autowiring does not.

Native-defined Regions

Spring Boot for Tanzu GemFire even auto-configures `GemfireTemplate` beans for Regions that have been defined with VMware GemFire native configuration metadata, such as `cache.xml`.

Consider the following VMware GemFire native `cache.xml`:

Example 6. Client `cache.xml`

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

    <region name="Example" refid="LOCAL"/>

</cache>
```

Further, consider the following Spring configuration:

Example 7. VMware GemFire Configuration

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

Spring Boot for Tanzu GemFire auto-configures a `GemfireTemplate` bean named `exampleTemplate` after the `Example` Region defined in `cache.xml`. You can inject this template as you would any other Spring-managed bean:

Example 8. Injecting the `GemfireTemplate`

```
@Service
class ExampleService {

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

}
```

The rules described earlier apply when multiple Regions are present.

Template Creation Rules

Fortunately, Spring Boot for Tanzu GemFire is careful not to create a `GemfireTemplate` bean for a Region if a template by the same name already exists.

For example, consider the following configuration:

Example 9. VMware GemFire Configuration

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

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

Further, consider the following example:

Example 10. Customer class

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

Because you explicitly defined and declared the `customersTemplate` bean, Spring Boot for Tanzu GemFire does not automatically create a template for the `Customers` Region. This applies regardless of how the Region was created, whether by using `@EnableEntityDefinedRegions`, `@EnableCachingDefinedRegions`, explicitly declaring Regions, or natively defining Regions.

Even if you name the template differently from the Region for which the template was configured, Spring Boot for Tanzu GemFire conserves resources and does 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 the `Customers` Region.

With the following configuration, Spring Boot for Tanzu GemFire is still careful not to create the template:

Example 11. VMware GemFire Configuration

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

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

Spring Boot for Tanzu GemFire identifies that your `vipCustomersTemplate` is the template used with the `Customers` Region, and Spring Boot for Tanzu GemFire does not create the `customersTemplate` bean,

which would result in two `GemfireTemplate` beans for the same Region.

Note: The name of your Spring bean defined in Java configuration is the name of the method if the Spring bean is not explicitly named by using the `name` attribute or the `value` attribute of the `@Bean` annotation.

Spring Data Repositories

This topic explains how to use Spring Data repositories with Spring Boot for VMware Tanzu GemFire.

Using Spring Data Repositories with VMware GemFire makes short work of data access operations when you use VMware GemFire as your System of Record (SoR) to persist your application's state.

Spring Data Repositories provide a convenient and powerful way to define basic CRUD and simple query data access operations by specifying the contract of those data access operations in a Java interface.

Spring Boot for Tanzu GemFire auto-configures the Spring Data for VMware GemFire Repository extension when either is declared on your application's classpath. You need not do anything special to enable it. You can start coding your application-specific Repository interfaces.

The following example defines a `Customer` class to model customers and map it to the VMware GemFire `Customers` Region by using the Spring Data for VMware GemFire `@Region` mapping annotation:

Example 1. `Customer` entity class

```
package example.app.crm.model;

@Region("Customers")
class Customer {

    @Id
    private Long id;

    private String name;

}
```

The following example shows how to declare your Repository for `Customers`:

Example 2. `CustomerRepository` for persisting and accessing `Customers`

```
package example.app.crm.repo;

interface CustomerRepository extends CrudRepository<Customer, Long> {

    List<Customer> findByLastNameLikeOrderByLastNameDescFirstNameAsc(String customerLa
stNameWildcard);

}
```

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

Example 3. Inject and use the `CustomerRepository`

```
package example.app;

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

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

    @Bean
    ApplicationRunner runner(CustomerRepository customerRepository) {

        return args -> {
            // Matches Williams, Wilson, etc.
            List<Customer> customers =
                    customerRepository.findByLastNameLikeOrderByLastNameDescFirstNameA
sc("Wil%");

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

For more detail, see Spring Data Commons' Repositories abstraction and Spring Data for VMware GemFire's Repositories extension.

Function Implementations and Executions

This topic discusses function implementation and execution in Spring Boot for VMware Tanzu GemFire, and using VMware GemFire in a Spring context for distributed computing use cases.

Background

Distributed computing, particularly in conjunction with data access and mutation operations, is a very effective and efficient use of clustered computing resources. This is similar to MapReduce.

A naively conceived query returning potentially hundreds of thousands (or even millions) of rows of data in a result set 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 more quickly. This typically involves intelligently organizing the data using various partitioning, or *sharding*, strategies to uniformly balance the data set across the cluster.

VMware GemFire addresses this very important application concern in its Function execution framework.

Spring Data for VMware GemFire builds on this function execution framework by letting developers execute VMware GemFire functions with a simple POJO-based annotation configuration model.

Taking this a step further, Spring Boot for Tanzu GemFire auto-configures and enables function execution out-of-the-box. Therefore, you can immediately begin invoking functions without having to worry about all the necessary plumbing to begin with.

Applying Functions

Earlier, when we talked about caching, we described a `FinancialLoanApplicationService` class that could process eligibility when someone (represented by a `Person` object) applied for a financial loan.

This can be a very resource intensive and expensive operation, since it might involve collecting credit and employment history, gathering information on outstanding loans, and so on. We applied caching to avoid having to recompute or redetermine eligibility every time a loan office may want to review the decision with the customer.

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 can implement an `EligibilityDeterminationFunction` class by using Spring Data for VMware GemFire:

Example 1. Function execution

Create a function with the ID “determineEligibility” and deploy it to your cluster.

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

  EligibilityDecision determineEligibility(Person person, Timespan timespan);

}
```

We can then inject an instance of the `EligibilityDeterminationExecution` interface into our `FinancialLoanApplicationService`, as we would any other object or Spring bean:

Example 2. Function use

```
@Service
class FinancialLoanApplicationService {

    private EligibilityDeterminationExecution execution;

    public FinancialLoanApplicationService(EligibilityDeterminationExecution executio
n) {
        this.execution = execution;
    }

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

As with caching, no additional configuration is required to enable and find your application Function implementations and executions. You can simply build and run. Spring Boot for Tanzu GemFire handles the rest.

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

Continuous Query

This topic discusses continuous querying with Spring Boot for VMware Tanzu GemFire.

Some applications must process a stream of events as they happen and intelligently react in (near) real-time to the countless changes in the data over time. Those applications need frameworks that can make processing a stream of events as they happen as easy as possible.

Spring Boot for Tanzu GemFire does just that, without users having to perform any complex setup or configure any necessary infrastructure components to enable such functionality. Developers can define the criteria for the data of interest and implement a handler (listener) to process the stream of events as they occur.

Continuous Query (CQ) lets you easily define your criteria for the data you need. With CQ, you can express the criteria that match the data you need by specifying a query predicate. VMware GemFire implements the Object Query Language (OQL) for defining and executing queries. OQL resembles SQL and supports projections, query predicates, ordering, and aggregates. Also, 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 Tanzu GemFire combines the ease of identifying the data you need by using an OQL query statement with implementing the listener callback (handler) in one easy step.

For example, suppose you want to perform some follow-up action when a customer's financial loan application is either approved or denied.

First, the application model for our `EligibilityDecision` class might look something like the following:

Example 1. 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 an eligibility 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, when eligibility is processed and a decision has been made, either approved or denied, our application gets notified, and as an application developer, you are free to code your handler and respond to the event any way you like. Also, because our Continuous Query (CQ) handler class is a component (or a 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], which are used in (JMS) message listeners and handlers, except in Spring Boot for Tanzu GemFire, you need not do anything special to enable this functionality. You can declare the `@ContinuousQuery` annotation on any POJO method and go to work on other things.

Using Data

This topic discusses using data with Spring Boot for VMware Tanzu GemFire.

One of the most important tasks during development is ensuring your Spring Boot application handles data correctly. To verify the accuracy, integrity, and availability of your data, your application needs data with which to work.

For those of you already familiar with Spring Boot's support for SQL database initialization, the approach when using VMware GemFire should be easy to understand.

Spring Boot for Tanzu GemFire offers support to import data from JSON into VMware GemFire as PDX. Spring Boot for Tanzu GemFire offers support to export data as well. By default, data is imported and exported in JSON format.

Spring Boot for Tanzu GemFire does not provide an equivalent to Spring Boot's `schema.sql` file. The best way to define the data structures (the `Region` instances) that manage your data is with Spring Data for VMware GemFire's annotation-based configuration support for defining cache `Region` instances from your application's entity classes or indirectly from Spring and JSR-107 or JCache caching annotations.

Warning: While this feature works and many edge cases were thought through and tested thoroughly, there are still some limitations. The Spring team strongly recommends that this feature be used only for development and testing purposes.

You can import data into a `Region` by defining a JSON file that contain the JSON objects you wish to load. The JSON file must follow a predefined naming convention and be placed in the root of your application classpath:

`data-<regionName>.json`

Note: `<regionName>` refers to the lowercase "name" of the `Region`, as defined by `Region.getName()` (see VMware GemFire Java API Reference).

For example, if you have a `Region` named "Orders", you would create a JSON file called `data-orders.json` and place it in the root of your application classpath (for example, in `src/test/resources`).

Create JSON files for each `Region` that is implicitly defined (for example, by using `@EnableEntityDefinedRegions`) or explicitly defined (with `ClientRegionFactoryBean` in Java configuration) in your Spring Boot application configuration that you want to load with data.

The JSON file that contains JSON data for the "Orders" `Region` might appear as follows:

Example 1. `data-orders.json`

```
[{
  "@type": "example.app.pos.model.PurchaseOrder",
```

```
  "id": 1,
  "lineItems": [
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Apple iPad Pro",
        "price": 1499.00,
        "category": "SHOPPING"
      },
      "quantity": 1
    },
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Apple iPhone 11 Pro Max",
        "price": 1249.00,
        "category": "SHOPPING"
      },
      "quantity": 2
    }
  ]
}, {
  "@type": "example.app.pos.model.PurchaseOrder",
  "id": 2,
  "lineItems": [
    {
      "@type": "example.app.pos.model.LineItem",
      "product": {
        "@type": "example.app.pos.model.Product",
        "name": "Starbucks Vente Carmel Macchiato",
        "price": 5.49,
        "category": "SHOPPING"
      },
      "quantity": 1
    }
  ]
}]
```

The application entity classes that matches the JSON data from the JSON file might look something like the following listing:

Example 2. Point-of-Sale (POS) Application Domain Model Classes

```
@Region("Orders")
class PurchaseOrder {

    @Id
    Long id;

    List<LineItem> lineItems;

}

class LineItem {

    Product product;
    Integer quantity;
```

```
}

@Region("Products")
class Product {

    String name;
    Category category;
    BigDecimal price;

}
```

As the preceding listings show, the object model and corresponding JSON can be arbitrarily complex with a hierarchy of objects that have complex types.

JSON metadata

We want to draw your attention to a few other details contained in the object model and JSON.

The `@type` metadata field

First, we declared a `@type` JSON metadata field. This field does not map to any specific field or property of the application domain model class (such as `PurchaseOrder`). Rather, it tells the framework and VMware GemFire's JSON/PDX converter the type of object the JSON data would map to if you were to request an object (by calling `PdxInstance.getObject()`).

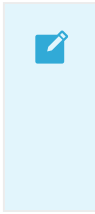
Consider the following example:

Example 3. Deserializing PDX as an Object

```
@Repository
class OrdersRepository {

    @Resource(name = "Orders")
    Region<Long, PurchaseOrder> orders;

    PurchaseOrder findBy(Long id) {

        Object value = this.orders.get(id);

        return value instanceof PurchaseOrder ? (PurchaseOrder) value
            : value instanceof PdxInstance ? (PurchaseOrder) ((PdxInstance) value).get
Object()
            : null;
    }
}
```

Basically, the `@type` JSON metadata field informs the `PdxInstance.getObject()` method about the type of Java object to which the JSON object maps. Otherwise, the `PdxInstance.getObject()` method would silently return a `PdxInstance`.

It is possible for VMware GemFire's PDX serialization framework to return a `PurchaseOrder` from `Region.get(key)` as well, but it depends on the value of PDX's `read-serialized`, cache-level configuration setting, among other factors.



Note: When JSON is imported into a `Region` as PDX, the `PdxInstance.getClassName()` does not refer to a valid Java class. It is `JSONFormatter.JSON_CLASSNAME`. As a result, `Region` data access operations, such as `Region.get(key)`, return a `PdxInstance` and not a Java object.

Tip	You may need to proxy `Region` read data access operations (such as `Region.get(key)`) by setting the Spring Boot for Tanzu GemFire property `spring.boot.data.gemfire.cache.region.advice.enabled` to `true`. When this property is set, `Region` instances are proxied to wrap a `PdxInstance` in a `PdxInstanceWrapper` to appropriately handle the `PdxInstance.getObject()` call in your application code.

The `id` field and the `@identifier` metadata field

Top-level objects in your JSON must have an identifier, such as an `id` field. This identifier is used as the identity and key of the object (or `PdxInstance`) when stored in the `Region` (for example, `Region.put(key, object)`).

You may have noticed that the JSON for the “Orders” `Region` shown earlier declared an `id` field as the identifier:

Example 4. PurchaseOrder identifier (“id”)

```
[{
  "@type": "example.app.pos.model.PurchaseOrder",
  "id": 1,
  ...
```

This follows the same convention used in Spring Data. Typically, Spring Data mapping infrastructure looks for a POJO field or property annotated with `@Id`. If no field or property is annotated with `@Id`, the framework falls back to searching for a field or property named `id`.

In Spring Data for VMware GemFire, this `@Id`-annotated or `id`-named field or property is used as the identifier and as the key for the object when storing it into a `Region`.

However, what happens when an object or entity does not have a surrogate ID defined? Perhaps the application domain model class is appropriately using natural identifiers, which is common in practice.

Consider a `Book` class defined as follows:

Example 5. Book class

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

    Author author;

    @Id
    ISBN isbn;

    LocalDate publishedDate;

    String title;
```

```
}
```

As declared in the `Book` class, the identifier for `Book` is its `ISBN`, since the `isbn` field was annotated with Spring Data's `@Id` mapping annotation. However, we cannot know this by searching for an `@Id` annotation in JSON.

You might be tempted to argue that if the `@type` metadata field is set, we would know the class type and could load the class definition to learn about the identifier. That is all fine until the class is not on the application classpath in the first place. This is one of the reasons why Spring Boot for Tanzu GemFire's JSON support serializes JSON to VMware GemFire's PDX format. There might not be a class definition, which would lead to a `NoClassDefFoundError` or `ClassNotFoundException`.

So, what then?

In this case, Spring Boot for Tanzu GemFire lets you declare the `@identifier` JSON metadata field to inform the framework what to use as the identifier for the object.

Consider the following example:

Example 6. Using "@identifer"

```
{
  "@type": "example.app.books.model.Book",
  "@identifier": "isbn",
  "author": {
    "id": 1,
    "name": "Josh Long"
  },
  "isbn": "978-1-449-374640-8",
  "publishedDate": "2017-08-01",
  "title": "Cloud Native Java"
}
```

The `@identifier` JSON metadata field informs the framework that the `isbn` field is the identifier for a `Book`.

Conditionally Importing Data

While the Spring team recommends that users should only use this feature when developing and testing their Spring Boot applications with VMware GemFire, you may still occasionally use this feature in production.

You might use this feature in production to preload a (REPLICATE) Region with reference data. Reference data is largely static, infrequently changing, and non-transactional. Preloading reference data is particularly useful when you want to warm the cache.

When you use this feature for development and testing purposes, you can put your `Region`-specific JSON files in `src/test/resources`. This ensures that the files are not included in your application artifact (such as a JAR or WAR) when built and deployed to production.

However, if you must use this feature to preload data in your production environment, you can still conditionally load data from JSON. To do so, configure the `spring.boot.data.gemfire.cache.data.import.active-profiles` property set to the Spring profiles that must be active for the import to take effect.

Consider the following example:

Example 7. Conditional Importing JSON

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.active-profiles=DEV, QA
```

For import to have an effect in this example, you must specifically set the `spring.profiles.active` property to one of the valid, `active-profiles` listed in the import property (such as `QA`). Only one needs to match.

Note: There are many ways to conditionally build application artifacts. You might prefer to handle this concern in your Gradle or Maven build.

Exporting Data

Certain data stored in your application's `Regions` may be sensitive or confidential, and keeping the data secure is of the utmost concern and priority. Therefore, exporting data is **disabled** by default.

However, if you use this feature for development and testing purposes, enabling the export capability may be useful to move data from one environment to another. For example, if your QA team finds a bug in the application that uses a particular data set, they can export the data and pass it back to the development team to import in their local development environment to help debug the issue.

To enable export, set the `spring.boot.data.gemfire.cache.data.export.enabled` property to `true`:

Example 8. Enable Export

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.export.enabled=true
```

Spring Boot for Tanzu GemFire is careful to export data to JSON in a format that VMware GemFire expects on import and includes things such as `@type` metadata fields.

Warning: The `@identifier` metadata field is not generated automatically. While it is possible for POJOs stored in a `Region` to include an `@identifier` metadata field when exported to JSON, it is not possible when the `Region` value is a `PdxInstance` that did not originate from JSON. In this case, you must manually ensure that the `PdxInstance` includes an `@identifier` metadata field before it is exported to JSON if necessary (for example, `Book.isbn`). This is only necessary if your entity classes do not declare an explicit identifier field, such as with the `@Id` mapping annotation, or do not have an `id` field. This scenario can also occur when interoperating with native clients that model the

application domain objects differently and then serialize the objects by using PDX, storing them in Regions on the server that are then later consumed by your Java-based, Spring Boot application.

Warning: You may need to set the `-Dgemfire.disableShutdownHook` JVM System property to `true` before your Spring Boot application starts up when using export. Unfortunately, this Java runtime shutdown hook is registered and enabled in VMware GemFire by default, which results in the cache and the Regions being closed before the Spring Boot for Tanzu GemFire Export functionality can export the data, thereby resulting in a `CacheClosedException`. Spring Boot for Tanzu GemFire makes a best effort to disable the VMware GemFire JVM shutdown hook when export is enabled, but it is at the mercy of the JVM `ClassLoader`, since VMware GemFire's JVM shutdown hook registration is declared in a `static` initializer.

Import/Export API Extensions

The API in Spring Boot for Tanzu GemFire for import and export functionality is separated into the following concerns:

- Data Format
- Resource Resolving
- Resource Reading
- Resource Writing

By breaking each of these functions apart into separate concerns, a developer can customize each aspect of the import and export functions.

For example, you could import XML from the filesystem and then export JSON to a REST-based Web Service. By default, Spring Boot for Tanzu GemFire imports JSON from the classpath and exports JSON to the filesystem.

However, not all environments expose a filesystem, such as cloud environments like the Tanzu Platform for Cloud Foundry. Therefore, giving users control over each aspect of the import and export processes is essential for performing the functions in any environment.

Data Format

The primary interface to import data into a `Region` is `CacheDataImporter`.

`CacheDataImporter` is a `@FunctionalInterface` that extends Spring's `BeanPostProcessor` interface to trigger the import of data after the `Region` has been initialized.

The interface is defined as follows:

Example 9. `CacheDataImporter`

```
interface CacheDataImporter extends BeanPostProcessor {

    Region importInto(Region region);

}
```

You can code the `importInto(:Region)` method to handle any data format (JSON, XML, and others) you prefer. Register a bean that implements the `CacheDataImporter` interface in the Spring container, and the importer does its job.

On the flip side, the primary interface to export data from a `Region` is the `CacheDataExporter`.

`CacheDataExporter` is a `@FunctionalInterface` that extends Spring's `DestructionAwareBeanPostProcessor` interface to trigger the export of data before the `Region` is destroyed.

The interface is defined as follows:

Example 10. `CacheDataExporter`

```
interface CacheDataExporter extends DestructionAwareBeanPostProcessor {

    Region exportFrom(Region region);
}
```

You can code the `exportFrom(:Region)` method to handle any data format (JSON, XML, and others) you prefer. Register a bean implementing the `CacheDataExporter` interface in the Spring container, and the exporter does its job.

For convenience, when you want to implement both import and export functionality, Spring Boot for Tanzu GemFire provides the `CacheDataImporterExporter` interface, which extends both `CacheDataImporter` and `CacheDataExporter`:

Example 11. `CacheDataImporterExporter`

```
interface CacheDataImporterExporter extends CacheDataExporter, CacheDataImporter { }
```

For added support, Spring Boot for Tanzu GemFire also provides the `AbstractCacheDataImporterExporter` abstract base class to simplify the implementation of your importer/exporter.

Lifecycle Management

Sometimes, it is necessary to precisely control when data is imported or exported.

This is especially true on import, since different `Region` instances may be collocated or tied together through a cache callback, such as a `CacheListener`. In these cases, the other `Region` may need to exist before the import on the dependent `Region` proceeds, particularly if the dependencies were loosely defined.

In all cases, Spring Boot for Tanzu GemFire provides the `LifecycleAwareCacheDataImporterExporter` class to wrap your `CacheDataImporterExporter` implementation. This class implements Spring's `SmartLifecycle` interface.

By implementing the `SmartLifecycle` interface, you can control in which `phase` of the Spring container the import occurs. Spring Boot for Tanzu GemFire also exposes two more properties to control the lifecycle:

Example 12. Lifecycle Management Properties

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.lifecycle=[EAGER|LAZY]
spring.boot.data.gemfire.cache.data.import.phase=1000000
```

`EAGER` acts immediately, after the `Region` is initialized (the default behavior). `LAZY` delays the import until the `start()` method is called, which is invoked according to the `phase`, thereby ordering the import relative to the other lifecycle-aware components that are registered in the Spring container.

The following example shows how to make your `CacheDataImporterExporter` lifecycle-aware:

```
@Configuration
class MyApplicationConfiguration {

    @Bean
    CacheDataImporterExporter importerExporter() {
        return new LifecycleAwareCacheDataImporterExporter(new MyCacheDataImporterExpo
rter());
    }
}
```

Resource Resolution

Resolving resources used for import and export results in the creation of a Spring `Resource` handle.

Resource resolution is a vital step to qualifying a resource, especially if the resource requires special logic or permissions to access it. In this case, specific `Resource` handles can be returned and used by the reader and writer of the `Resource` as appropriate for import or export operation.

Spring Boot for Tanzu GemFire encapsulates the algorithm for resolving `Resources` in the `ResourceResolver` (Strategy) interface:

Example 13. ResourceResolver

```
@FunctionalInterface
interface ResourceResolver {

    Optional<Resource> resolve(String location);

    default Resource required(String location) {
        // ...
    }
}
```

Additionally, Spring Boot for Tanzu GemFire provides the `ImportResourceResolver` and `ExportResourceResolver` marker interfaces and the `AbstractImportResourceResolver` and `AbstractExportResourceResolver` abstract base classes for implementing the resource resolution logic used by both import and export operations.

If you wish to customize the resolution of `Resources` used for import or export, your `CacheDataImporterExporter` implementation can extend the `ResourceCapableCacheDataImporterExporter` abstract base class, which provides the aforementioned interfaces and base classes.

As stated earlier, Spring Boot for Tanzu GemFire resolves resources on import from the classpath and resources on export to the filesystem.

You can customize this behavior by providing an implementation of `ImportResourceResolver`, `ExportResourceResolver`, or both interfaces and declare instances as beans in the Spring context:

Example 14. Import and Export ResourceResolver beans

```
@Configuration
class MyApplicationConfiguration {

    @Bean
    ImportResourceResolver importResourceResolver() {
        return new MyImportResourceResolver();
    }

    @Bean
    ExportResourceResolver exportResourceResolver() {
        return new MyExportResourceResolver();
    }
}
```

Tip	If you need to customize the resource resolution process for each location (or `Region`) on import or export, you can use the Composite software design pattern.

Customize Default Resource Resolution

If you are content with the provided defaults but want to target specific locations on the classpath or filesystem used by the import or export, Spring Boot for Tanzu GemFire additionally provides the following properties:

Example 15. Import/Export Resource Location Properties

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.resource.location=...
spring.boot.data.gemfire.cache.data.export.resource.location=...
```

The properties accept any valid resource string, as specified in the Spring documentation (see **Table 10. Resource strings**).

This means that, even though import defaults from the classpath, you can change the location from classpath to filesystem, or even network (for example, https://) by changing the prefix (or protocol).

Import/export resource location properties can refer to other properties through property placeholders, but Spring Boot for Tanzu GemFire further lets you use SpEL inside the property values.

Consider the following example:

Example 16. Using SpEL

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.import.resource.location=\
  https://#{#env['user.name']}:#{someBean.lookupPassword(#env['user.name'])}@#{host}:#
{port}/cache/#{#regionName}/data/import
```

In this case, the import resource location refers to a rather sophisticated resource string by using a complex SpEL expression.

Spring Boot for Tanzu GemFire populates the SpEL `EvaluationContext` with three sources of information:

- Access to the Spring `BeanFactory`
- Access to the Spring `Environment`
- Access to the current `Region`

Simple Java System properties or environment variables can be accessed with the following expression:

```
#{propertyName}
```

You can access more complex property names (including properties that use dot notation, such as the `user.home` Java System property), directly from the `Environment` by using map style syntax as follows:

```
#{#env['property.name']}
```

The `#env` variable is set in the SpEL `EvaluationContext` to the Spring `Environment`.

Because the SpEL `EvaluationContext` is evaluated with the Spring `ApplicationContext` as the root object, you also have access to the beans declared and registered in the Spring container and can invoke methods on them, as shown earlier with `someBean.lookupPassword(..)`. `someBean` must be the name of the bean as declared and registered in the Spring container.

Note: Be careful when accessing beans declared in the Spring container with SpEL, particularly when using `EAGER` import, as it may force those beans to be eagerly (or even prematurely) initialized.

Spring Boot for Tanzu GemFire also sets the `#regionName` variable in the `EvaluationContext` to the name of the `Region`, as determined by `Region.getName()` (see VMware GemFire Java API Reference), targeted for import and export.

This lets you not only change the location of the resource but also change the resource name (such as a filename).

Consider the following example:

Example 17. Using `#regionName`

```
# Spring Boot application.properties

spring.boot.data.gemfire.cache.data.export.resource.location=\
    file://#{#env['user.home']}/gemfire/cache/data/custom-filename-for-#{#regionName}.
json
```

Note: By default, the exported file is stored in the working directory (`System.getProperty("user.dir")`) of the Spring Boot application process.

Tip	See the Spring Framework documentation for more information about SpEL.

Reading and Writing Resources

The Spring `Resource` handle specifies tion of a resource, not how the resource is read or written. Even the Spring `ResourceLoader`, which is an interface for loading `Resources`, does not specifically read or write any content to the `Resource`.

Spring Boot for Tanzu GemFire separates these concerns into two interfaces: `ResourceReader` and `ResourceWriter`, respectively. The design follows the same pattern used by Java's `InputStream/OutputStream` and `Reader/Writer` classes in the `java.io` package.

The `ResourceReader` interfaces is defined as:

Example 18. ResourceReader

```
@FunctionalInterface
interface ResourceReader {

    byte[] read(Resource resource);

}
```

The `ResourceWriter` interfaces is defined as:

Example 19. ResourceWriter

```
@FunctionalInterface
interface ResourceWriter {

    void write(Resource resource, byte[] data);

}
```

Both interfaces provide additional methods to compose readers and writers, much like Java's `Consumer` and `Function` interfaces in the `java.util.function` package. If a particular reader or writer is used in a composition and is unable to handle the given `Resource`, it should throw a `UnhandledResourceException` to let the next reader or writer in the composition try to read from or write to the `Resource`.

The reader or writer are free to throw a `ResourceReadException` or `ResourceWriteException` to break the chain of reader and writer invocations in the composition.

To override the default export/import reader and writer used by Spring Boot for Tanzu GemFire, you can implement the `ResourceReader` or `ResourceWriter` interfaces as appropriate and declare instances of these classes as beans in the Spring container:

Example 20. Custom `ResourceReader` and `ResourceWriter` beans

```
@Configuration
class MyApplicationConfiguration {
```

```
    @Bean
    ResourceReader myResourceReader() {
        return new MyResourceReader()
            .thenReadFrom(new MyOtherResourceReader());
    }

    @Bean
    ResourceWriter myResourceWriter() {
        return new MyResourceWriter();
    }
}
```

Data Serialization with PDX

This topic discusses data serialization with PDX in Spring Boot for VMware Tanzu GemFire.

Anytime data is overflowed or persisted to disk, transferred between clients and servers, transferred between peers in a cluster or between different clusters in a multi-site WAN topology, all data stored in VMware 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`, refactoring hundreds or even thousands of class types to implement `java.io.Serializable` would be a tedious task just to store and manage those objects in VMware GemFire.

Additionally, it is not only your application domain object types you necessarily need to consider. If you used third-party libraries in your application domain model, any types referred to by your application domain object types stored in VMware GemFire must also be serializable. 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 metadata 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 VMware GemFire's PDX format. PDX stands for Portable Data Exchange and achieves four goals:

- Separates type metadata from the data itself, streamlining the bytes during transfer. VMware GemFire maintains a type registry that stores type metadata about the objects serialized with PDX.
- Supports versioning as your application domain types evolve. It is common to have old and new versions of the same application deployed to production, running simultaneously, sharing data, and possibly using different versions of the same domain types. PDX lets fields be added or removed while still preserving interoperability between old and new application clients without loss of data.
- Enables objects stored as PDX to be queried without being deserialized. Constant serialization and deserialization of data is a resource-intensive task that adds to the latency of each data request when redundancy is enabled. Since data is replicated across peers in the cluster to preserve High Availability (HA) and must be serialized to be transferred, keeping data serialized is more efficient when data is updated frequently, since it is likely the data will need to be transferred again in order to maintain consistency in the face of redundancy and availability.
- Enables interoperability between native language clients (such as C, C++ and C#) and Java language clients, with each being able to access the same data set regardless of where the data originated.

However, PDX does have limitations.

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 VMware GemFire's general Data Serialization handles Deltas, this is not achievable without deserializing the object, since it involves a method invocation, which defeats one of the key benefits of PDX: preserving format to avoid the cost of serialization and deserialization.

However, we think the benefits of using PDX outweigh the limitations and, therefore, have enabled PDX by default.

You don't need to do anything special. You can code your domain types and rest assured that objects of those domain types are properly serialized when overflowed and persisted to disk, transferred between clients and servers, transferred between peers in a cluster, and even when data is transferred over the network when you use VMware GemFire's multi-site WAN topology.

Example 1. EligibilityDecision is automatically serializable without implementing Java Serializable.

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

VMware GemFire supports the standard Java Serialization format. For more information, see Data Serialization in the VMware GemFire product documentation.

MappingPdxSerializer versus ReflectionBasedAutoSerializer

Spring Boot for Tanzu GemFire enables and uses Spring Data for VMware GemFire's `MappingPdxSerializer`to serialize your application domain objects with PDX.

The `MappingPdxSerializer` class offers several advantages above and beyond VMware GemFire's own `ReflectionBasedAutoSerializer` class (see VMware GemFire Java API Reference).

See VMware GemFire's User Guide for more details about the `ReflectionBasedAutoSerializer`.

The Spring Data for VMware GemFire `MappingPdxSerializer` class offers the following benefits and capabilities:

- PDX serialization is based on Spring Data's powerful mapping infrastructure and metadata.
- Includes support for both `includes` and `excludes` with first-class type filtering. Additionally, you can implement type filters by using Java's `java.util.function.Predicate` interface as opposed to the limited regex capabilities provided by VMware GemFire's `ReflectionBasedAutoSerializer` class. By default, `MappingPdxSerializer` excludes all types in the following packages: `java`, `org.apache.geode`, `org.springframework` and `com.gemstone.gemfire`.
- Handles transient object fields and properties when either Java's `transient` keyword or Spring Data's `@Transient` annotation is used.
- Handles read-only object properties.
- Automatically determines the identifier of your entities when you annotate the appropriate entity field or property with Spring Data's `@Id` annotation.

- Allows additional `o.a.g.pdx.PdxSerializers` to be registered to customize the serialization of nested entity/object field and property types.

The support for `includes` and `excludes` deserves special attention, since the `MappingPdxSerializer` excludes all Java, Spring, and VMware GemFire types, by default. However, what happens when you need to serialize one of those types?

For example, suppose you need to serialize objects of type `java.security.Principal`. Then you can override the excludes by registering an `include` type filter:

```
package example.app;

import java.security.Principal;

@SpringBootApplication
@EnablePdx(serializerBeanName = "myCustomMappingPdxSerializer")
class SpringBootGemFireClientCacheApplication {

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

    @Bean
    MappingPdxSerializer myCustomMappingPdxSerializer() {

        MappingPdxSerializer customMappingPdxSerializer =
            MappingPdxSerializer.newMappingPdxSerializer();

        customMappingPdxSerializer.setIncludeTypeFilters(
            type -> Principal.class.isAssignableFrom(type));

        return customMappingPdxSerializer;
    }
}
```

Tip	Normally, you need not explicitly declare Spring Data for VMware GemFire's `@EnablePdx` annotation to enable and configure PDX. However, if you want to override auto-configuration, as we have demonstrated above, you must do this.

Logging

This topic explains how to configure Spring Boot for VMware Tanzu GemFire logging.

To get log output from VMware GemFire, VMware GemFire requires a logging provider declared on your Spring Boot application classpath. Consequently, this also means the old VMware GemFire `Properties` (such as `log-level`) no longer have any effect, regardless of whether the property is specified in `gemfire.properties`, in Spring Boot `application.properties`, or even as a JVM System Property (`-Dgemfire.log-level`).

Tip: See VMware GemFire's documentation for a complete list of valid `Properties`, including the properties used to configure logging.

Configure VMware GemFire Logging

Three things are required to get VMware GemFire to log output:

1. You must declare a logging provider (such as Logback, or Log4j) on your Spring Boot application classpath.
2. **(optional)** You can declare an adapter (a bridge JAR) between Log4j and your logging provider if your declared logging provider is not Apache Log4j.

   For example, if you use the SLF4J API to log output from your Spring Boot application and use Logback as your logging provider or implementation, you must include the `org.apache.logging.log4j.log4j-to-slf4j` adapter or bridge JAR as well.

   Internally, VMware GemFire uses the Apache Log4j API to log output from GemFire components. Therefore, you must bridge Log4j to any other logging provider (such as Logback) that is not Log4j (`log4j-core`). If you use Log4j as your logging provider, you need not declare an adapter or bridge JAR on your Spring Boot application classpath.
3. Finally, you must supply logging provider configuration to configure Loggers, Appenders, log levels, and other details.

   For example, when you use Logback, you must provide a `logback.xml` configuration file on your Spring Boot application classpath or in the filesystem. Alternatively, you can use other means to configure your logging provider and get VMware GemFire to log output.

If you declare Spring Boot's own `org.springframework.boot:spring-boot-starter-logging` on your application classpath, it covers steps 1 and 2 above.

The `spring-boot-starter-logging` dependency declares Logback as the logging provider and automatically adapts (bridges) `java.util.logging` (JUL) and Apache Log4j to SLF4J. However, you still need to supply logging provider configuration (such as a `logback.xml` file for Logback) to configure logging not only for your Spring Boot application but for VMware GemFire as well.

Note: If no user-specified logging configuration is supplied, Logback will apply default configuration using the `BasicConfigurator`. See Logbackfor complete details.

Spring Boot for Tanzu GemFire has simplified the setup of VMware GemFire logging. You need only declare the `com.vmware.gemfire:spring-boot-logging...` dependency on your Spring Boot application classpath.

Unlike VMware GemFire's default Log4j XML configuration file (`log4j2.xml` from `geode-log4j`), Spring Boot for Tanzu GemFire's provided `logback.xml` configuration file is properly parameterized, letting you adjust log levels, add Appenders as well as adjust other logging settings.

In addition, Spring Boot for Tanzu GemFire's provided Logback configuration uses templates so that you can compose your own logging configuration while still including snippets from Spring Boot for Tanzu GemFire's provided logging configuration, such as Loggers and Appenders.

Configuring Log Levels

One of the most common logging tasks is to adjust the log level of one or more Loggers or the ROOT Logger. However, you may want to only adjust the log level for specific components of your Spring Boot application, such as for VMware GemFire, by setting the log level for only the Logger that logs VMware GemFire events.

Spring Boot for Tanzu GemFire's Logback configuration defines three Loggers to control the log output from VMware GemFire:

Example VMware GemFire Loggers by Name

```
<comfiguration>
  <logger name="com.gemstone.gemfire" level="${spring.boot.data.gemfire.log.level:-INF
O}"/>
  <logger name="org.apache.geode" level="${spring.boot.data.gemfire.log.level:-INFO}"/
>
  <logger name="org.jgroups" level="${spring.boot.data.gemfire.jgroups.log.level:-WAR
N}"/>
</comfiguration>
```

The `com.gemstone.gemfire` Logger covers old GemFire components that are still present in VMware GemFire for backwards compatibility. By default, it logs output at `INFO`. This Logger's use should be mostly unnecessary.

The `org.apache.geode` Logger is the primary Logger used to control log output from all VMware GemFire components during the runtime operation of VMware GemFire. By default, it logs output at `INFO`.

The `org.jgroups` Logger is used to log output from VMware GemFire's message distribution and membership system. VMware GemFire uses JGroups for membership and message distribution between peer members (nodes) in the cluster (distributed system). By default, JGroups logs output at `WARN`.

You can configure the log level for the `com.gemstone.gemfire` and `org.apache.geode` Loggers by setting the `spring.boot.data.gemfire.log.level` property. You can independently configure the `org.jgroups` Logger by setting the `spring.boot.data.gemfire.jgroups.log.level` property.

You can set the Spring Boot for Tanzu GemFire logging properties on the command line as JVM System properties when you run your Spring Boot application:

Example - Setting the log-level from the CLI

```
$ java -classpath ...:/path/to/MySpringBootApplication.jar -Dspring.boot.data.gemfire.
log.level=DEBUG
    package.to.MySpringBootApplicationClass
```

Note: Setting JVM System properties by using `$ java -jar MySpringBootApplication.jar -Dspring.boot.data.gemfire.log.level=DEBUG` is not supported by the Java Runtime Environment (JRE).

Alternatively, you can configure and control VMware GemFire logging in Spring Boot `application.properties`:

Example - Setting the log-level in Spring Boot `application.properties`

```
spring.boot.data.gemfire.log.level=DEBUG
```

For backwards compatibility, Spring Boot for Tanzu GemFire additionally supports the Spring Data for VMware GemFire logging properties as well, by using either of the following properties:

Example - Setting log-level using Spring Data for VMware GemFire Properties

```
spring.data.gemfire.cache.log-level=DEBUG
spring.data.gemfire.logging.level=DEBUG
```

Composing Logging Configuration

As mentioned earlier, Spring Boot for Tanzu GemFire lets you compose your own logging configuration from Spring Boot for Tanzu GemFire's default Logback configuration metadata.

Spring Boot for Tanzu GemFire conveniently bundles the Properties, Loggers, and Appenders from Spring Boot for Tanzu GemFire's logging starter into several template files that you can include in your own custom Logback XML configuration file.

The Logback configuration template files are broken down into:

- `org/springframework/geode/logging/slf4j/logback/properties-include.xml`
- `org/springframework/geode/logging/slf4j/logback/loggers-include.xml`
- `org/springframework/geode/logging/slf4j/logback/appenders-include.xml`

Warning: As of Spring Boot for Tanzu GemFire **3.0**,the logback-include.xml file was removed.

The `properties-include.xml` defines Logback "*local*" scoped properties or variables common to Spring Boot for Tanzu GemFire's configuration of VMware GemFire logging.

Example - properties-include.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<included>

    <property name="SPRING_BOOT_LOG_CHARSET" value="${SPRING_BOOT_LOG_CHARSET:-${file.
encoding:-UTF-8}}"/>
    <property name="SPRING_BOOT_LOG_PATTERN" value="${SPRING_BOOT_LOG_PATTERN:-%d %5p
%40.40c:%4L - %msg%n}"/>
    <property name="APACHE_GEODE_LOG_CHARSET" value="${APACHE_GEODE_LOG_CHARSET:-${fil
e.encoding:-UTF-8}}"/>
```

```
    <property name="APACHE_GEODE_LOG_PATTERN" value="${APACHE_GEODE_LOG_PATTERN:-[%lev
el{lowerCase=true} %date{yyyy/MM/dd HH:mm:ss.SSS z} &lt;%thread&gt;] %message%n%throwa
ble%n}"/>

</included>
```

The `loggers-include.xml` file defines the `Loggers` used to log output from VMware GemFire components.

Example - loggers-include.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<included>

    <logger name="com.gemstone.gemfire" level="${spring.boot.data.gemfire.log.level:-I
NFO}" additivity="false">
        <appender-ref ref="${spring.geode.logging.appender-ref:-CONSOLE}"/>
        <appender-ref ref="delegate"/>
    </logger>

    <logger name="org.apache.geode" level="${spring.boot.data.gemfire.log.level:-INF
O}" additivity="false">
        <appender-ref ref="${spring.geode.logging.appender-ref:-CONSOLE}"/>
        <appender-ref ref="delegate"/>
    </logger>

    <logger name="org.jgroups" level="${spring.boot.data.gemfire.jgroups.log.level:-WA
RN}" additivity="false">
        <appender-ref ref="${spring.geode.logging.appender-ref:-CONSOLE}"/>
        <appender-ref ref="delegate"/>
    </logger>

</included>
```

The `appenders-include.xml` file defines Appenders to send the log output to. If Spring Boot is on the application classpath, then Spring Boot logging configuration will define the “CONSOLE” `Appender`, otherwise, Spring Boot for Tanzu GemFire will provide a default definition.

The “geode” `Appender` defines the VMware GemFire logging pattern as seen in VMware GemFire’s Log4j configuration.

Example - appenders-include.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<included>

    <if condition='property("bootPresent").equals("false")'>
        <then>
            <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
                <encoder>
                    <charset>${SPRING_BOOT_LOG_CHARSET}</charset>
                    <pattern>${SPRING_BOOT_LOG_PATTERN}</pattern>
                </encoder>
            </appender>
        </then>
    </if>

    <appender name="delegate" class="org.springframework.geode.logging.slf4j.logback.D
elegatingAppender"/>
```

```
    <appender name="geode" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <charset>${APACHE_GEODE_LOG_CHARSET}</charset>
            <pattern>${APACHE_GEODE_LOG_PATTERN}</pattern>
        </encoder>
    </appender>

</included>
```

Then you can include any of Spring Boot for Tanzu GemFire'S Logback configuration metadata files as needed in your application-specific Logback XML configuration file, as follows:

Example - application-specific logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <include resource="org/springframework/geode/logging/slf4j/logback/properties-incl
ude.xml"/>
    <include resource="org/springframework/geode/logging/slf4j/logback/appender-includ
e.xml"/>

    <logger name="org.apache.geode" level="INFO" additivity="false">
        <appender-ref ref="geode"/>
    </logger>

    <root level="${logback.root.log.level:-INFO}">
        <appender-ref ref="CONSOLE"/>
        <appender-ref ref="delegate"/>
    </root>

</configuration>
```

Customizing Logging Configuration

It is now possible to customize the configuration of VMware GemFire logging using properties defined in a `spring-geode-logging.properties` file included on the Spring Boot application classpath.

Any of the properties defined in `org/springframework/geode/logging/slf4j/logback/properties-include.xml` (shown above), such as `APACHE_GEODE_LOG_PATTERN`, or the `spring.geode.logging.appender-ref` property, can be set.

For instance, and by default, VMware GemFire components log output using the Spring Boot log pattern. However, if you prefer the fine-grained details of VMware GemFire's logging behavior, you can change the `Appender` used by the VMware GemFire `Logger's` to use the pre-defined "geode" `Appender` instead. Set the `spring-geode.logging.appender-ref` property to "geode" in a `spring-geode-logging.properties` file on your Spring Boot application classpath, as follows:

Example - spring-geode-logging.properties

```
# spring-geode-logging.properties
spring.geode.logging.appender-ref=geode
```

Alternatively, if you want to configure the log output of your entire Spring Boot application, including log output from all VMware GemFire components, then you can set the `SPRING_BOOT_LOG_PATTERN` property, or Spring Boot's `CONSOLE_LOG_PATTERN` property, in `spring-geode-logging.properties`, as follows:

Example - spring-geode-logging.properties

```
# spring-geode-logging.properties
CONSOLE_LOG_PATTERN=TEST - %msg%n
```

Note: The `spring-geode-logging.properties` file is only recognized when the `spring-boot-logging...` module is used.

SLF4J and Logback API Support

Spring Boot for Tanzu GemFire provides additional support when working with the SLF4J and Logback APIs. This support is available when you declare the `com.vmware.boot:spring-boot-logging...` dependency on your Spring Boot application classpath.

One of the main supporting classes from the `spring-boot-logging...` is the `org.springframework.geode.logging.slf4j.logback.LogbackSupport` class. This class provides methods to:

- Resolve a reference to the Logback `LoggingContext`.
- Resolve the SLF4J ROOT `Logger` as a Logback `Logger`.
- Look up `Appenders` by name and required type.
- Add or remove `Appenders` to `Loggers`.
- Reset the state of the Logback logging system, which can prove to be most useful during testing.

`LogbackSupport` can even suppress the auto-configuration of Logback performed by Spring Boot on startup, which is another useful utility during automated testing.

In addition to the `LogbackSupport` class, Spring Boot for Tanzu GemFire also provides some custom Logback `Appenders`.

CompositeAppender

The `org.springframework.geode.logging.slf4j.logback.CompositeAppender` class is an implementation of the Logback `Appender` interface and the Composite software design pattern.

`CompositeAppender` lets developers compose multiple Appenders and use them as if they were a single `Appender`.

For example, you could compose both the Logback `ConsoleAppender` and `FileAppender` into one `Appender`:

Example - Composing multiple `Appenders`

```
class LoggingConfiguration {

  Appender<ILoggingEvent> compositeAppender() {

    ConsoleAppender<ILoggingEvent> consoleAppender = new ConsoleAppender<>();
```

```
    FileAppender<ILoggingEvent> fileAppender = new FileAppender<>();

    Appender<ILoggingEvent> compositeAppender = CompositeAppender.compose(consoleAppen
der, fileAppender);

    return compositeAppender;
  }
}

// do something with the compositeAppender
```

You could then add the `CompositeAppender` to a named `Logger`:

Example - Register `CompositeAppender` on “named” `Logger`

```
void registerAppenderOnLogger() {

    Logger namedLogger = LoggerFactory.getLogger("loggerName");

    LogbackSupport.toLogbackLogger(namedLogger)
      .ifPresent(it -> LogbackSupport.addAppender(it, compositeAppender()));
}
```

In this case, the named `Logger` logs events (or log messages) to both the console and file Appenders.

You can compose an array or `Iterable` of `Appenders` by using either the `CompositeAppender.compose(:Appender<T>[])` method or the `CompositeAppender.compose(:Iterable<Appender<T>>)` method.

DelegatingAppender

The `org.springframework.geode.logging.slf4j.logback.DelegatingAppender` is a pass-through Logback `Appender` implementation that wraps another Logback `Appender` or collection of `Appenders`, such as the `ConsoleAppender`, a `FileAppender`, a `SocketAppender`, or others. By default, the `DelegatingAppender` delegates to the `NOPAppender`, thereby doing no actual work.

By default, Spring Boot for Tanzu GemFire registers the `org.springframework.geode.logging.slf4j.logback.DelegatingAppender` with the ROOT `Logger`, which can be useful for testing purposes.

With a reference to a `DelegatingAppender`, you can add any `Appender` (even a `CompositeAppender`) as the delegate:

Example - Add `ConsoleAppender` as the “delegate” for the `DelegatingAppender`

```
class LoggerConfiguration {

  void setupDelegation() {

    ConsoleAppender consoleAppender = new ConsoleAppender();

    LogbackSupport.resolveLoggerContext().ifPresent(consoleAppender::setContext);

    consoleAppender.setImmediateFlush(true);
    consoleAppender.start();
```

```
    LogbackSupport.resolveRootLogger()
      .flatMap(LogbackSupport::toLogbackLogger)
      .flatMap(rootLogger -> LogbackSupport.resolveAppender(rootLogger,
        LogbackSupport.DELEGATE_APPENDER_NAME, DelegatingAppender.class))
      .ifPresent(delegateAppender -> delegateAppender.setAppender(consoleAppender));
  }
}
```

StringAppender

The `org.springframework.geode.logging.slf4j.logback.StringAppender` stores a log message in-memory, appended to a `String`.

The `StringAppender` is useful for testing purposes. For instance, you can use the `StringAppender` to assert that a `Logger` used by certain application components logged messages at the appropriately configured log level while other log messages were not logged.

Consider the following example:

Example - `StringAppender` in Action

```
class ApplicationComponent {

    private final Logger logger = LoggerFactory.getLogger(getClass());

    public void someMethod() {
        logger.debug("Some debug message");
        // ...
    }

    public void someOtherMethod() {
        logger.info("Some info message");
    }
}

// Assuming the ApplicationComponent Logger was configured with log-level 'INFO', the
n...
class ApplicationComponentUnitTests {

    private final ApplicationComponent applicationComponent = new ApplicationComponent
();

    private final Logger logger = LoggerFactory.getLogger(ApplicationComponent.class);

    private StringAppender stringAppender;

    @Before
    public void setup() {

        LogbackSupport.toLogbackLogger(logger)
            .map(Logger::getLevel)
            .ifPresent(level -> assertThat(level).isEqualTo(Level.INFO));

        stringAppender = new StringAppender.Builder()
            .applyTo(logger)
            .build();
    }
```

```
    @Test
    public void someMethodDoesNotLogDebugMessage() {

        applicationComponent.someMethod();

        assertThat(stringAppender.getLogOutput).doesNotContain("Some debug message");
    }

    @Test
    public void someOtherMethodLogsInfoMessage() {

        applicationComponent.someOtherMethod();

        assertThat(stringAppender.getLogOutput()).contains("Some info message");
    }
}
```

There are many other uses for the `StringAppender` and you can use it safely in a multi-Threaded context by calling `StringAppender.Builder.useSynchronization()`.

When combined with other Spring Boot for Tanzu GemFire provided `Appenders` in conjunction with the `LogbackSupport` class, you have a lot of power both in application code and in your tests.

Security

This topic discusses security configurations for Spring Boot for VMware Tanzu GemFire, and includes authentication and authorization, as well as Transport Layer Security (TLS) using SSL.

Tip	See the corresponding sample guide and security in the `spring-boot-for-vmware-gemfire` repository in GitHub to see Spring Boot Security for VMware GemFire in action.

Authentication and Authorization

VMware GemFire employs username- and password-based authentication and role-based authorization to secure your client to server data exchanges and operations.

Spring Data for VMware GemFire provides first-class support for VMware GemFire's Security framework, which is based on the `SecurityManager` interface (see VMware GemFire Java API Reference). Additionally, VMware GemFire's Security framework is integrated with Apache Shiro.

When you use Spring Boot for Tanzu GemFire, which builds Spring Data for VMware GemFire, it makes short work of enabling auth in your clients.

Auth for Clients

When servers in an VMware GemFire cluster have been configured with authentication and authorization enabled, clients must authenticate when connecting.

Spring Boot for Tanzu GemFire makes this easy, regardless of whether you run your Spring Boot `ClientCache` applications in a local, non-managed environment or run in a cloud-managed environment.

Non-Managed Auth for Clients

To enable auth for clients that connect to a secure VMware GemFire cluster, you need only set a username and password in Spring Boot `application.properties`:

Example 1. Spring Boot `application.properties` for the client

```
# Spring Boot client application.properties

spring.data.gemfire.security.username = jdoe
spring.data.gemfire.security.password = p@55w0rd
```

Spring Boot for Tanzu GemFire handles the rest.

Managed Auth for Clients

See Tanzu Platform for Cloud Foundry for additional details on user authentication and authorization.

Transport Layer Security using SSL

Securing data in motion is also essential to the integrity of your Spring [Boot] applications.

For instance, it would not do much good to send usernames and passwords over plain text socket connections between your clients and servers nor to send other sensitive data over those same connections.

Therefore, VMware GemFire supports SSL between clients and servers, between JMX clients (such as Gfsh) and the Manager, between HTTP clients when you use the Developer REST API or Pulse, between peers in the cluster, and when you use the WAN Gateway to connect multiple sites (clusters).

Spring Data for VMware GemFire provides first-class support for configuring and enabling SSL as well. Spring Boot makes it even easier to configure and enable SSL, especially during development.

VMware GemFire requires certain properties to be configured. These properties translate to the appropriate `javax.net.ssl.*` properties required by the JRE to create secure socket connections by using JSSE.

However, ensuring that you have set all the required SSL properties correctly is an error-prone and tedious task. Therefore, Spring Boot for Tanzu GemFire applies some basic conventions for you.

You can create a `trusted.keystore` as a JKS-based `KeyStore` file and place it in one of three well-known locations:

- In your application JAR file at the root of the classpath.
- In your Spring Boot application's working directory.
- 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 one of these three well-known locations, Spring Boot for Tanzu GemFire automatically configures your client to use SSL socket connections.

Example 2. 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 one of the default, well-known locations searched by Spring Boot:

Example 3. Accessing a secure `trusted.keystore` by location

```
# 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/truststor
e.jks
spring.data.gemfire.security.ssl.truststore.password = trustorePassword
```

See the Spring Data for VMware GemFire `EnableSsl` annotation for all the configuration attributes and the corresponding properties expressed in `application.properties`.

Testing

For testing purposes, Spring Boot for Tanzu GemFire provides a test implementation of VMware GemFire's `SecurityManager` interface (see VMware GemFire Java API Reference), which expects the password to match the username (case-sensitive) when authenticating.

By default, all operations are authorized.

To match the expectations of Spring Boot for Tanzu GemFire's `TestSecurityManager`, Spring Boot for Tanzu GemFire additionally provides a test implementation of VMware GemFire's `AuthInitialize` interface, which supplies matching credentials for both the username and password.

VMware GemFire API Extensions

This topic discusses using GemFire API extensions with Spring Boot for VMware Tanzu GemFire.

When using the Spring programming model and abstractions, it should not be necessary to use VMware GemFire APIs at all, for example, when using the Spring Cache Abstraction for caching or the Spring Data Repository abstraction for DAO development. There are many more examples.

`SimpleCacheResolver`

In some cases, it is necessary to acquire a reference to the cache instance in your application components at runtime. For example, you might want to create a temporary `Region` on the fly to aggregate data for analysis.

In a Spring context, the cache instance created by the framework is a managed bean in the Spring container. You can inject a reference to the *Singleton* cache bean into any other managed application component:

Example 1. Autowired Cache Reference using Dependency Injection (DI)

```
@Service
class CacheMonitoringService {

    @Autowired
    ClientCache clientCache;

    // use the clientCache object reference to monitor the cache as necessary

}
```

However, in cases where your application component or class is not managed by Spring and you need a reference to the cache instance at runtime, Spring Boot for Tanzu GemFire provides the abstract `org.springframework.geode.cache.SimpleCacheResolver` class.

Example 2. `SimpleCacheResolver` API

```
package org.springframework.geode.cache;

abstract class SimpleCacheResolver {

    <T extends ClientCache> T require() {...}

    <T extends ClientCache> Optional<T> resolve() {...}
}
```

`SimpleCacheResolver` adheres to SOLID OO Principles. This class is abstract and extensible so that you can change the algorithm used to resolve client cache instances as well as mock its methods in unit tests.

`resolve()` resolves a reference to the client cache if present and returns `Optional.EMPTY` if not.

`require()` returns a non-`Optional` reference to a cache instance and throws an `IllegalStateException` if a cache is not present.

`CacheUtils`

Under the hood, `SimpleCacheResolver` delegates some of its functions to the `CacheUtils` abstract utility class, which provides additional, convenient capabilities when you use a cache. Such as extracting all the values from a Region.

To extract all the values stored in a Region, call `CacheUtils.collectValues(:Region<?, T>)`. This method returns a `Collection<T>` that contains all the values stored in the given `Region`. The method is smart and knows how to handle the `Region` appropriately. This distinction is important, since client `PROXY` Regions store no values.

Warning: VMware advises caution when you get all values from a Region. While getting filtered reference values from a non-transactional, reference data only [`REPLICATE`] Region is useful, getting all values from a transactional, [`PARTITION`] Region can prove quite detrimental, especially in production. Getting all values from a Region can be useful during testing.

PDX

VMware GemFire's PDX serialization framework is a convenient and efficient way to serialize data and a good alternative to Java serialization.

Note that it is not possible to use PDX in local-only mode without a server, since the PDX type registry is only available and managed on servers in a cluster.

`PdxInstanceBuilder`

In such cases, Spring Boot for Tanzu GemFire conveniently provides the `PdxInstanceBuilder` class, appropriately named after the Builder software design pattern. The `PdxInstanceBuilder` also offers a fluent API for constructing `PdxInstances`:

Example 3. `PdxInstanceBuilder` API

```
class PdxInstanceBuilder {

    PdxInstanceFactory copy(PdxInstance pdx) {...}

    Factory from(Object target) {...}

}
```

For example, you could serialize an application domain object as PDX bytes with the following code:

Example 4. Serializing an Object to PDX

```
@Component
class CustomerSerializer {
```

```
    PdxInstance serialize(Customer customer) {

        return new PdxInstanceBuilder()
            .from(customer)
            .create();
    }
}
```

You could then modify the `PdxInstance` by copying from the original:

Example 5. Copy `PdxInstance`

```
@Component
class CustomerDecorator {

    @Autowired
    CustomerSerializer serializer;

    PdxInstance decorate(Customer customer) {

        PdxInstance pdxCustomer = serializer.serialize(customer);

        return PdxInstanceBuilder.create()
            .copy(pdxCustomer)
            .writeBoolean("vip", isImportant(customer))
            .create();
    }
}
```

`PdxInstanceWrapper`

Spring Boot for Tanzu GemFire also provides the `PdxInstanceWrapper` class to wrap an existing `PdxInstance` in order to provide more control during the conversion from PDX to JSON and from JSON back into a POJO. Specifically, the wrapper gives you more control over the configuration of Jackson's `ObjectMapper`.

Because the `ObjectMapper` constructed by VMware GemFire's own `PdxInstance` implementation (`PdxInstanceImpl`) is not configurable and `PdxInstance` is not extensible, the `getObject()` method can fail when converting the JSON generated from PDX back into a POJO for some application domain model types.

The following example wraps an existing `PdxInstance`:

Example 6. Wrapping an existing `PdxInstance`

```
PdxInstanceWrapper wrapper = PdxInstanceWrapper.from(pdxInstance);
```

For all operations on `PdxInstance` except `getObject()`, the wrapper delegates to the underlying `PdxInstance` method implementation called by the user.

In addition to the decorated `getObject()` method, the `PdxInstanceWrapper` provides a thorough implementation of the `toString()` method. The state of the `PdxInstance` is output in a JSON-like `String`.

Finally, the `PdxInstanceWrapper` class adds a `getIdentifier()` method. Rather than put the burden on the user to have to iterate the field names of the `PdxInstance` to determine whether a field is the identity field and then call `getField(name)` with the field name to get the ID (value) — assuming an identity field was marked in the first place — the `PdxInstanceWrapper` class provides the `getIdentifier()` method to return the ID of the `PdxInstance` directly.

The `getIdentifier()` method is smart in that it first iterates the fields of the `PdxInstance`, asking each field if it is the identity field. If no field was marked as the identity field, the algorithm searches for a field named `id`. If no field with the name `id` exists, the algorithm searches for a metadata field called `@identifier`, which refers to the field that is the identity field of the `PdxInstance`.

The `@identifier` metadata field is useful in cases where the `PdxInstance` originated from JSON and the application domain object uses a natural identifier, rather than a surrogate ID, such as `Book.isbn`.

Note: VMware GemFire's `JSONFormatter` class is not capable of marking the identity field of a `PdxInstance` originating from JSON.

Warning: It is not currently possible to implement the `PdxInstance` interface and store instances of this type as a value in a Region. VMware GemFire assumes all `PdxInstance` objects are an implementation created by VMware GemFire itself (that is, `PdxInstanceImpl`), which has a tight coupling to the PDX type registry. An `Exception` is thrown if you try to store instances of your own `PdxInstance` implementation.

`ObjectPdxInstanceAdapter`

In rare cases, you may need to treat an `Object` as a `PdxInstance`, depending on the context without incurring the overhead of serializing an `Object` to PDX. For such cases, Spring Boot for Tanzu GemFire offers the `ObjectPdxInstanceAdapter` class.

This might be true when calling a method with a parameter expecting an argument of, or returning an instance of, type `PdxInstance`, particularly when VMware GemFire's `read-serialized` PDX configuration property is set to `true` and only an object is available in the current context.

Spring Boot for Tanzu GemFire's `ObjectPdxInstanceAdapter` class uses Spring's `BeanWrapper` class along with Java's introspection and reflection functionality to adapt the given `Object` and access it with the full `PdxInstance` API. This includes the use of the `WritablePdxInstance` API, obtained from `PdxInstance.createWriter()`, to modify the underlying `Object` as well.

Like the `PdxInstanceWrapper` class, `ObjectPdxInstanceAdapter` contains special logic to resolve the identity field and ID of the `PdxInstance`, including consideration for Spring Data's `@Id` mapping annotation, which can be introspected in this case, given that the underlying `Object` backing the `PdxInstance` is a POJO.

The `ObjectPdxInstanceAdapter.getObject()` method returns the wrapped `Object` used to construct the `ObjectPdxInstanceAdapter` and is, therefore, automatically deserializable, as determined by the `PdxInstance.isDeseriable()` method, which always returns `true`.

You can adapt any `Object` as a `PdxInstance`:

Example 7. Adapt an `Object` as a `PdxInstance`

```
class OfflineObjectToPdxInstanceConverter {

    @NonNull PdxInstance convert(@NonNull Object target) {
        return ObjectPdxInstanceAdapter.from(target);
    }
}
```

Once the Adapter is created, you can use it to access data on the underlying `Object`.

Consider the following example of a `Customer` class:

Example 8. `Customer` class

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

    @Id
    private Long id;

    String name;

    // constructors, getters and setters omitted

}
```

Then you can access an instance of `Customer` by using the `PdxInstance` API:

Example 9. Accessing an `Object` using the `PdxInstance` API

```
class ObjectPdxInstanceAdapterTest {

    @Test
    public void getAndSetObjectProperties() {

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

        PdxInstance adapter = ObjectPdxInstanceAdapter.from(jonDoe);

        assertThat(jonDoe.getName()).isEqualTo("Jon Doe");
        assertThat(adapter.getField("name")).isEqualTo("Jon Doe");

        adapter.createWriter().setField("name", "Jane Doe");

        assertThat(adapter.getField("name")).isEqualTo("Jane Doe");
        assertThat(jonDoe.getName()).isEqualTo("Jane Doe");
    }
}
```

Spring Boot Actuator

Spring Boot for VMware Tanzu GemFire adds Spring Boot Actuator support and dedicated `HealthIndicators` for VMware GemFire. Equally, the provided `HealthIndicators` even work with Tanzu Cache (which is backed by GemFire) when you push your Spring Boot applications using GemFire to VMware Tanzu Platform for Cloud Foundry.

Spring Boot `HealthIndicators` provide details about the runtime operation and behavior of your VMware GemFire-based Spring Boot applications. For instance, by querying the right `HealthIndicator` endpoint, you can get the current hit/miss count for your `Region.get(key)` data access operations.

In addition to vital health information, Spring Boot for Tanzu GemFire provides basic, pre-runtime configuration metadata about the VMware 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 configuration, XML, and so on.

The provided Spring Boot `HealthIndicators` include:

`Regions`, `DiskStores`, `ContinuousQuery`, and connection `Pools`.

The following sections give a brief overview of all the available Spring Boot `HealthIndicators` provided for VMware GemFire.

To see a Spring Boot Actuator example, see the corresponding sample in Spring Boot Actuator for VMware GemFire and actuator in GitHub.

GeodeCacheHealthIndicator

`GeodeCacheHealthIndicator` provides essential details about the (single) cache instance and the underlying `DistributedSystem`, the `DistributedMember` and configuration details of the `ResourceManager`.

The `ClientCache`creates instances of the `DistributedSystem` and `DistributedMember` objects, respectively.

Each object has the following configuration metadata and health details:

Table 1. Cache Details

Name	Description
geode.cache.name	Name of the member in the distributed system.
geode.cache.closed	Determines whether the cache has been closed.

Name	Description
geode.cache.cancel-in-progress	Indicates whether cancellation of operations is in progress.

Table 1. Cache Details

Table 2. DistributedMember Details

Name	Description
geode.distributed-member.id	`DistributedMember` identifier (used in logs internally).
geode.distributed-member.name	Name of the member in the distributed system.
geode.distributed-members.groups	Configured groups to which the member belongs.
geode.distributed-members.host	Name of the machine on which the member is running.
geode.distributed-members.process-id	Identifier of the JVM process (PID).

Table 2. DistributedMember Details

Table 3. DistributedSystem Details

Name	Description
geode.distributed-system.connected	Indicates whether the member is currently connected to the cluster.
geode.distributed-system.member-count	Total number of members in the cluster (1 for clients).
geode.distributed-system.reconnecting	Indicates whether the member is in a reconnecting state, which happens when a network partition occurs and the member gets disconnected from the cluster.
geode.distributed-system.properties-location	Location of the standard configuration properties.
geode.distributed-system.security-properties-location	Location of the security configuration properties.

Table 3. DistributedSystem Details

Table 4. ResourceManager Details

Name	Description
geode.resource-manager.critical-heap-percentage	Percentage of heap at which the cache is in danger of becoming inoperable.
geode.resource-manager.eviction-heap-percentage	Percentage of heap at which eviction begins on Regions configured with a heap LRU eviction policy.

Table 4. ResourceManager Details

GeodeRegionsHealthIndicator

`GeodeRegionsHealthIndicator` provides details about all the configured and known `Regions` in the cache. Details include all `LOCAL`, `PROXY`, and `CACHING_PROXY Regions`.

The following table describes the essential details and basic performance metrics:

Table 5. Region Details

Name	Description
geode.cache.regions.<name>.cloning-enabled	Whether Region values are cloned on read (for example, `cloning-enabled` is `true` when cache transactions are used to prevent in-place modifications).
geode.cache.regions.<name>.data-policy	Policy used to manage data in the Region.
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.
geode.cache.regions.<name>.key-constraint	Type constraint for Region keys.
geode.cache.regions.<name>.pool-name	If this Region is a client Region, this property determines the configured connection `Pool`. (NOTE: Regions can have and use dedicated `Pools` for their data access operations.)
geode.cache.regions.<name>.pool-name	Determines the `Scope` of the Region, which plays a factor in the Region's consistency-level, as it pertains to acknowledgements for writes.
geode.cache.regions.<name>.value-constraint	Type constraint for Region values.

Table 5. Region Details

When statistics are enabled, the metadata below is available.

Note: For example, you can enable statistics when you use `@EnableStatistics`.

Table 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, indicates the last time it was accessed with `Region.get(key)`.
geode.cache.regions.<name>.statistics.last-modified-time	For an entry, indicates the time when 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.

Table 7. Region Statistic Details

GeodeDiskStoresHealthIndicator

The `GeodeDiskStoresHealthIndicator` provides details about the configured `DiskStores` in the system or application. Remember, `DiskStores` are used to overflow and persist data to disk, including type metadata tracked by PDX when the values in the Regions have been serialized with PDX and the Regions are persistent.

Most of the tracked health information pertains to configuration:

Table 10. DiskStore Details

Name	Description
geode.disk-store.<name>.allow-force-compaction	Indicates whether manual compaction of the DiskStore is allowed.
geode.disk-store.<name>.auto-compact	Indicates whether compaction occurs automatically.

Name	Description
geode.disk-store.<name>.compaction-threshold	Percentage at which the oplog becomes compactible.
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 that stores the disk files.
geode.disk-store.<name>.disk-usage-critical-percentage	Critical threshold of disk usage proportional to the total disk volume.
geode.disk-store.<name>.disk-usage-warning-percentage	Warning threshold of disk usage proportional to the total disk volume.
geode.disk-store.<name>.max-oplog-size	Maximum size (MB) allowed for a single oplog file.
geode.disk-store.<name>.queue-size	Size of the queue used to batch writes that are 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 a distributed system.
geode.disk-store.<name>.write-buffer-size	Size the of write buffer the DiskStore uses to write data to disk.

Table 10. DiskStore Details

GeodeContinuousQueriesHealthIndicator

`GeodeContinuousQueriesHealthIndicator` provides details about registered client Continuous Queries (CQs). CQs let client applications receive automatic notification about events that satisfy some criteria. That criteria can be easily expressed by using the predicate of an OQL query (for example, `SELECT * FROM /Customers c WHERE c.age > 21`). When data is inserted or updated and the data matches the criteria specified in the OQL query predicate (data of interests), an event is sent to the registered client.

The following details are covered for CQs by name:

Table 11. Continuous Query (CQ) Details

Name	Description
geode.continuous-query.<name>.oql-query-string	OQL query constituting the CQ.
geode.continuous-query.<name>.closed	Indicates whether the CQ has been closed.
geode.continuous-query.<name>.closing	Indicates whether the CQ is in the process of closing.
geode.continuous-query.<name>.durable	Indicates whether the CQ events are 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.

Table 11. Continuous Query (CQ) Details

In addition, the following CQ query and statistical data is covered:

Table 12. Continuous Query (CQ), Query Details

Name	Description
geode.continuous-query.<name>.query.number-of-executions	Total number of times the query has been executed.
geode.continuous-query.<name>.query.total-execution-time	Total amount of time (ns) spent executing the query.

Table 12. Continuous Query (CQ), Query Details

Table 13. Continuous Query(CQ), Statistic Details

Name	Description
geode.continuous-query.<name>.statistics.number-of-deletes	Number of delete events qualified by this CQ.

Name	Description
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.

Table 13. Continuous Query(CQ), Statistic Details

The VMware GemFire Continuous Query system is also tracked with the following additional details on the client:

Table 14. Continuous Query (CQ), Additional 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).
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).

Table 14. Continuous Query (CQ), Additional Statistic Details

GeodePoolsHealthIndicator

`GeodePoolsHealthIndicator` provides details about all the configured client connection `Pools`. This `HealthIndicator` primarily provides configuration metadata for all the configured `Pools`.

The following details are covered:

Table 15. Pool Details

Name	Description
geode.pool.count	Total number of client connection pools.
geode.pool.<name>.destroyed	Indicates whether the pool has been destroyed.
geode.pool.<name>.free-connection-timeout	Configured amount of time to wait for a free connection from the Pool.
geode.pool.<name>.idle-timeout	The amount of time to wait before closing unused, idle connections, not exceeding the configured number of minimum required connections.
geode.pool.<name>.load-conditioning-interval	How frequently the Pool checks to see whether 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 the server for this durable client Pool at the time it (re)connected to the server.
geode.pool.<name>.ping-interval	How often to ping the servers to verify they are still alive.

Name	Description
geode.pool.<name>.pr-single-hop-enabled	Whether the client acquires a direct connection to the server.
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 a timeout or an exception.
geode.pool.<name>.server-group	All servers must belong to the same group, and this value sets the name of that group.
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 (TTL) period (ms) for subscription events the client has received from the server.
geode.pool.<name>.subscription-redundancy	Redundancy level for this Pool's server-to-client subscriptions, which is used to ensure clients do not miss potentially important events.
geode.pool.<name>.thread-local-connections	Thread local connection policy for this Pool.

Table 15. Pool Details

Spring Session

This topic discusses using Spring Session for VMware GemFire with Spring Boot for VMware Tanzu GemFire.

This topic discusses auto-configuration of Spring Session for VMware 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 and provide 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`.

Spring Boot for Tanzu GemFire provides auto-configuration support to configure VMware GemFire as the session management provider and store when Spring Session for VMware GemFire is on your Spring Boot application's classpath.

Tip: See the corresponding sample guideand code to see Spring Session for VMware GemFire in action.

Configuration

You do not need to do anything special to use VMware GemFire as a Spring Session provider implementation, managing the (HTTP) session state of your Spring Boot application.

To do so, declare the Spring Boot Session for GemFire dependency in your Spring Boot application Maven POM (shown here) or Gradle build file:

Example - Maven dependency declaration

```
<dependency>
  <groupId>com.vmware.gemfire</groupId>
  <artifactId>spring-boot-session-3.3-gemfire-10.1</artifactId>
  <version>2.0.0</version>
</dependency>
```

After declaring the required Spring Session dependency, you can begin your Spring Boot application as you normally would:

Example - Spring Boot Application

```
@SpringBootApplication
public class MySpringBootApplication {

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

```
  // ...
}
```

You can then create application-specific Spring Web MVC `Controllers` to interact with the `HttpSession` as needed by your application:

Example - Spring Boot Application `Controller` using `HttpSession`

```
@Controller
class MyApplicationController {

  @GetMapping("...")
  public String processGet(HttpSession session) {
    // interact with HttpSession
  }
}
```

The `HttpSession` is replaced by a Spring managed `Session` that is stored in VMware GemFire.

Note: Spring Boot Session for Tanzu GemFire, using Spring Boot's Autoconfiguration, enables subscriptions on GemFire's `DEFAULT` client pool. Due to performance reasons, it is advisable not to use subscriptions with Spring Boot Session for Tanzu GemFire. To disable the autoconfiguration one should use `@EnableAutoConfiguration(exclude = EnableSubscriptionConfiguration.class)` to disable the autoconfiguration OR create a separate pool for the Spring Session to use that does not enable subscriptions.

Custom Configuration

By default, Spring Boot for Tanzu GemFire applies reasonable and sensible defaults when configuring VMware GemFire as the provider in Spring Session.

For instance, by default, Spring Boot for Tanzu GemFire sets the session expiration timeout to 30 minutes. It also uses a `ClientRegionShortcut.PROXY` as the data management policy for the VMware GemFire client Region that managing the (HTTP) session state.

However, what if the defaults are not sufficient for your application requirements?

In that case, see the next section.

Custom Configuration using Properties

Spring Session for VMware GemFire publishes configuration properties for each of the various Spring Session configuration options when you use VMware GemFire as the (HTTP) session state management provider.

You can specify any of these properties in Spring Boot `application.properties` to adjust Spring Session's configuration when using VMware GemFire.

In addition to the properties provided in and by Spring Session for VMware GemFire, Spring Boot for Tanzu GemFire also recognizes and respects the `spring.session.timeout` property and the `server.servlet.session.timeout` property, as discussed the Spring Boot documentation.

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 VMware GemFire also provides the `SpringSessionGemFireConfigurer` callback interface, which you can declare in your Spring `ApplicationContext` to programmatically control the configuration of Spring Session when you use VMware GemFire.

The `SpringSessionGemFireConfigurer`, when declared in the Spring `ApplicationContext`, takes precedence over any of the Spring Session (for VMware GemFire) configuration properties and effectively overrides them when both are present.

Using Spring Session with GemFire for VMs

Whether you use Spring Session in a Spring Boot, GemFire `ClientCache` application to connect to a standalone, externally managed cluster of GemFire servers or to connect to a cluster of servers in a GemFire service instance managed by a Tanzu Platform for Cloud Foundry environment, the setup is the same.

Spring Session for GemFire expects there to be a cache Region in the cluster that can store and manage (HTTP) session state when your Spring Boot application is a `ClientCache` application in the client/server topology.

By default, the cache Region used to store and manage (HTTP) session state is called `ClusteredSpringSessions`.

We recommend that you configure the cache Region name by using the well-known and documented property in Spring Boot `application.properties`:

Example - Using properties

```
spring.session.data.gemfire.session.region.name=MySessions
```

Alternatively, you can set the name of the cache Region used to store and manage (HTTP) session state by explicitly declaring the `@EnableGemFireHttpSession` annotation on your main `@SpringBootApplication` class:

Example - Using `@EnableGemfireHttpSession

```
@SpringBootApplication
@EnableGemFireHttpSession(regionName = "MySessions")
class MySpringBootSpringSessionApplication {
    // ...
}
```

Once you decide on the cache Region name used to store and manage (HTTP) sessions, you must create the cache Region in the cluster somehow.

On the client, doing so is simple, since Spring Boot for Tanzu GemFire's auto-configuration automatically creates the client `PROXY` Region that is used to send and receive (HTTP) session state between the client

and server for you when either Spring Session is on the application classpath (for example, `spring-boot-session-3.0-gemfire-10.0`) or you explicitly declare the `@EnableGemFireHttpSession` annotation on your main `@SpringBootApplication` class.

On the server side, you can manually create the cache Region by using Gfsh:

Example - Create the Sessions Region using Gfsh

```
gfsh> create region --name=MySessions --type=PARTITION --entry-idle-time-expiration=18
00
        --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 (session object) is `invalidated`.

Note: Session expiration is managed by the Expiration Policy set on the cache Region that is used to store session state. The Servlet container's (HTTP) session expiration configuration is not used, since Spring Session replaces the Servlet container's session management capabilities with its own, and Spring Session delegates this behavior to the individual providers, such as VMware GemFire.

Example - Using Gfsh to Alter Region

```
gfsh> alter region --name=MySessions --entry-idle-time-expiration=1800
        --entry-idle-time-expiration-action=INVALIDATE
```

Your Spring Boot `ClientCache` application, configured with Spring Session in a client/server topology, can now store and manage user (HTTP) session state in the cluster. This setup works for externally managed GemFire clusters and GemFire running in a Tanzu Platform for Cloud Foundry environment.

VMware Tanzu Platform for Cloud Foundry

VMware Tanzu GemFire for Tanzu Platform for Cloud Foundry is a managed version of GemFire designed to run on the Tanzu Platform for Cloud Foundry. When deployed in cloud environments such as AWS, Azure, or GCP, GemFire for Cloud Foundry offers several advantages over managing standalone GemFire clusters. It takes care of many infrastructure and operational concerns, allowing you to focus on your application rather than the underlying management tasks.

Running a Spring Boot Application as a Specific User

By default, Spring Boot applications run as a `cluster_operator` role-based user in Tanzu Platform for Cloud Foundry when bound to a GemFire for Cloud Foundry service instance. This role has full system privileges, including comprehensive control over the GemFire service instance.

Running as a Different User

Not all applications need full privileges. To run a Spring Boot application with restricted permissions, you must create a user with limited access. This must be done on the Tanzu Platform for Cloud Foundry. Refer to the VMware GemFire for Tanzu Platform for Cloud Foundry security documentation for more information on configuring users with assigned roles and permissions.

Configuring a Spring Boot Application to Run as a Specific User

After creating a restricted user, configure your Spring Boot application to use this user by setting the appropriate property in the `application.properties` file:

```
# Spring Boot application.properties for Tanzu Platform for Cloud Foundry using GemFir
e
spring.data.gemfire.security.username=guest
```

This property specifies the runtime user for your application when connecting to an externally managed GemFire cluster. If the username is invalid, an `IllegalStateException` is thrown. Use Spring profiles to configure the application to run with different users depending on the environment.

Overriding Authentication Auto-configuration

In externally managed environments, you must explicitly set a username and password:

Example: Overriding Security Authentication Auto-configuration:

```
# Spring Boot application.properties
spring.data.gemfire.security.username=MyUser
spring.data.gemfire.security.password=MyPassword
```

Targeting Specific GemFire Service Instances

You can provision and bind multiple GemFire service instances to your Spring Boot application. However, Spring Boot for Tanzu GemFire only auto-configures one service instance. To target a specific instance, set the following property:

Example: Targeting a specific GemFire service instance:

```
# Spring Boot application.properties
spring.boot.data.gemfire.cloud.cloudfoundry.service.cloudcache.name=gemFireServiceInst
anceTwo
```

If the specified instance does not exist, an `IllegalStateException` is thrown.

Using Multiple GemFire Service Instances

To use multiple GemFire service instances, configure multiple connection `Pools`:

Example: Multiple GemFire Service Instance Configuration:

```
    @Configuration
    @EnablePools(pools = {
      @EnablePool(name = "One"),
      @EnablePool(name = "Two"),
      ...
      @EnablePool(name = "GemFireN")
    })
    class GemFireConfiguration {
      // ...
    }
```

Externalize the configuration for the declared `Pools` in `application.properties`:

Example: Configuring Locator-based Pool connections:

```
# Spring Boot application.properties
spring.data.gemfire.pool.one.locators=gemFireOneHost1[port1], gemFireOneHost2[port2],
..., gemFireOneHostN[portN]
spring.data.gemfire.pool.two.locators=gemFireTwoHost1[port1], gemFireTwoHost2[port2],
..., gemFireTwoHostN[portN]
```

Assign a Pool to a client Region:

Example: Assigning a Pool to a client Region:

```
@Configuration
class GemFireConfiguration {

  @Bean("Example")
  ClientRegionFactoryBean exampleRegion(GemFireCache gemfireCache,
      @Qualifier("Two") Pool poolTwo) {

    ClientRegionFactoryBean exampleRegion = new ClientRegionFactoryBean();

    exampleRegion.setCache(gemfireCache);
    exampleRegion.setPool(poolTwo);
    exampleRegion.setShortcut(ClientRegionShortcut.PROXY);
```

```
    return exampleRegion;
  }
}
```

Configure as many Pools and client Regions as needed. The `Pool` determines the GemFire service instance and cluster where the data for the client Region resides.

Docker

This topic discusses using Docker with Spring Boot for VMware Tanzu GemFire.

The state of modern software application development is moving towards containerization. Containers offer a controlled environment to predictably build (compile, configure and package), run, and manage your applications in a reliable and repeatable manner, regardless of context. In many situations, the intrinsic benefit of using containers is obvious.

Understandably, Docker's popularity took off like wildfire, given its highly powerful and simplified model for creating, using and managing containers to run packaged applications.

Docker's ecosystem is also impressive, with the advent of Testcontainers and Spring Boot's dedicated support to create packaged Spring Boot applications in Docker images that are then later run in a Docker container.

Tip	See also Deploying to Containers to learn more.

VMware GemFire can also run in a controlled, containerized environment. The goal of this chapter is to get you started running VMware GemFire in a container and interfacing to a containerized VMware GemFire cluster from your Spring Boot, VMware GemFire client applications.

This chapter does not cover how to run your Spring Boot, VMware GemFire client applications in a container, since that is already covered by Spring Boot (see the Spring Boot documentation for Docker images and container deployment, along with Docker's documentation). Instead, our focus is on how to connect to a VMware GemFire cluster in a container from a Spring Boot, VMware GemFire client application, regardless of whether the application runs in a container or not.

First, set up your containerized VMware GemFire cluster and make sure to map ports between the Docker container and the host system, exposing well-known ports used by VMware GemFire server-side cluster processes, such as Locators and CacheServers:

Table 1. VMware GemFire Ports

Process	Port
HTTP	7070
Locator	10334
Manager	1099
Server	40404

Table 1. VMware GemFire Ports

Spring Boot, VMware GemFire Client Application Explained

The Spring Boot, VMware GemFire `ClientCache` application we use to connect to our VMware GemFire cluster that runs in the Docker container appears as follows:

Example 1. Spring Boot, VMware GemFire Docker client application

```
@SpringBootApplication
@EnableClusterAware
@EnableEntityDefinedRegions(basePackageClasses = Customer.class)
@UseMemberName("SpringBootGemFireDockerClientCacheApplication")
public class SpringBootGemFireDockerClientCacheApplication {

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

    @Bean
    @SuppressWarnings("unused")
    ApplicationRunner runner(GemFireCache cache, CustomerRepository customerRepositor
y) {

        return args -> {

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

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

            log("Saving Customer [%s]...%n", jonDoe);

            jonDoe = customerRepository.save(jonDoe);

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

            log("Querying for Customer [SELECT * FROM /Customers WHERE name LIKE '%
s']...%n", "%Doe");

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

            assertThat(queriedJonDoe).isEqualTo(jonDoe);

            log("Customer was [%s]%n", queriedJonDoe);
        };
    }

    private void assertClientCacheAndConfigureMappingPdxSerializer(GemFireCache cache)
{

        assertThat(cache).isNotNull();
        assertThat(cache.getName())
            .isEqualTo(SpringBootGemFireDockerClientCacheApplication.class.getSimpleNa
me());
        assertThat(cache.getPdxSerializer()).isInstanceOf(MappingPdxSerializer.class);
```

```
        MappingPdxSerializer serializer = (MappingPdxSerializer) cache.getPdxSerialize
r();

        serializer.setIncludeTypeFilters(type -> Optional.ofNullable(type)
            .map(Class::getPackage)
            .map(Package::getName)
            .filter(packageName -> packageName.startsWith(this.getClass().getPackage
().getName()))
            .isPresent());
    }

    private void log(String message, Object... args) {
        System.err.printf(message, args);
        System.err.flush();
    }
}
```

Our `Customer` application domain model object type is defined as:

Example 2. `Customer` class

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

    @Id
    private Long id;

    private String name;

}
```

Also, we define a Spring Data CRUD Repository to persist and access `Customers` stored in the `/Customers` Region:

Example 3. `CustomerRepository` interface

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

    Customer findByNameLike(String name);

}
```

Our main class is annotated with `@SpringBootApplication`, making it be a proper Spring Boot application.

We additionally annotate the main class with Spring Boot for Tanzu GemFire's `@EnableClusterAware` annotation to automatically detect the VMware GemFire cluster that runs in the Docker container.

The application requires that a Region called "Customers", as defined by the `@Region` mapping annotation on the `Customer` application domain model class, exists on the servers in the cluster, to store `Customer` data.

We use the Spring Data for VMware GemFire `@EnableEntityDefinedRegions` annotation to define the matching client `PROXY` "Customers" Region.

Optionally, we have also annotated our main class with Spring Boot for Tanzu GemFire's `@UseMemberName` annotation to give the `ClientCache` a name, which we assert in the `assertClientCacheAndConfigureMappingPdxSerializer(:ClientCache)` method.

The primary work performed by this application is done in the Spring Boot `ApplicationRunner` bean definition. We create a `Customer` instance (`Jon Doe`), save it to the “Customers” Region managed by the server in the cluster, and then query for `Jon Doe` using OQL, asserting that the result is equal to what we expect.

We log the output from the application’s operations to see the application in action.

Running the Spring Boot, VMware GemFire client application

When you run the Spring Boot, VMware GemFire client application, you should see output similar to the following:

Example 4. Application log output

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/bin/java ...
    org.springframework.geode.docs.example.app.docker.SpringBootGemFireDockerClientCac
heApplication


  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)

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

Process finished with exit code 0
```

When we review the configuration of the cluster, we see that the `/Customers` Region was created when the application ran:

Example 5. /Customers Region Configuration

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


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

Non-Default Attributes Shared By Hosting Members

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

Our `/Customers` Region contains a value (`Jon Doe`), and we can verify this by running the following OQL Query with gfsh:

Example 6. Query the `/Customers` Region

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

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

Our application ran successfully.

Samples

This topic contains working examples that show how to use Spring Boot for VMware Tanzu GemFire effectively.

Some examples focus on specific use cases (such as (HTTP) session state caching), while other examples show how Spring Boot for Tanzu GemFire works under the hood, to give you a better understanding of what is actually happening and how to debug problems with your Spring Boot for Tanzu GemFire applications.

Table 1. Example Spring Boot applications using GemFire

Guide	Description	Source
Spring Boot Auto-Configuration for VMware GemFire	Explains what auto-configuration is provided by Spring Boot for Tanzu GemFire and what the auto-configuration does.	Spring Boot Auto-Configuration
Spring Boot Actuator for VMware GemFire	Explains how to use Spring Boot Actuator for VMware GemFire and how it works.	Spring Boot Actuator
Spring Boot Security for VMware GemFire	Explains how to configure auth and TLS with SSL when you use VMware GemFire in your Spring Boot applications.	Spring Boot Security
Look-Aside Caching with Spring's Cache Abstraction and VMware GemFire	Explains how to enable and use Spring's Cache Abstraction with VMware GemFire as the caching provider for look-aside caching.	Look-Aside Caching
Inline Caching with Spring's Cache Abstraction and VMware GemFire	Explains how to enable and use Spring's Cache Abstraction with VMware GemFire as the caching provider for inline caching. This sample builds on the look-aside caching sample.	Inline Caching
Near Caching with Spring's Cache Abstraction and VMware GemFire	Explains how to enable and use Spring's Cache Abstraction with VMware GemFire as the caching provider for near caching. This sample builds on the look-aside caching sample.	Near Caching

Guide	Description	Source
HTTP Session Caching with Spring Session and VMware GemFire	Explains how to enable and use Spring Session with VMware GemFire to manage HTTP session state.	HTTP Session Caching

Appendix

This topic lists additional resources to use when developing Spring Boot for VMware Tanzu GemFire applications.

- Auto-Configuration versus Annotation-Based Configuration
- Configuration Metadata Reference
- Deactivating Auto-Configuration

Auto-Configuration versus Annotation-Based Configuration

This topic discusses the Spring Boot for VMware Tanzu GemFire annotations you can or must use when developing VMware GemFire applications with Spring Boot.

See the complementary sample, Spring Boot Auto-configuration for VMware GemFire, which shows the auto-configuration provided by Spring Boot for Tanzu GemFire in action.

Background

To start, review the complete collection of available Spring Data for VMware GemFire annotations. These annotations are provided in the `org.springframework.data.gemfire.config.annotation` package. Most of the essential annotations begin with `@Enable…`.

By extension, Spring Boot for Tanzu GemFire builds on Spring Data for VMware GemFire's annotation-based configuration model to implement auto-configuration and apply Spring Boot's core concepts, such as "convention over configuration", letting VMware GemFire applications be built with Spring Boot reliably, quickly, and easily.

Spring Data for VMware GemFire provides this annotation-based configuration model to, first and foremost, give application developers "choice" when building Spring applications with VMware GemFire. Spring Data for VMware GemFire makes no assumptions about what application developers are trying to create and fails fast anytime the configuration is ambiguous, giving users immediate feedback.

Second, Spring Data for VMware GemFire's annotations were meant to get application developers up and running quickly and reliably with ease. Spring Data for VMware GemFire accomplishes this by applying sensible defaults so that application developers need not know, or even have to learn, all the intricate configuration details and tooling provided by VMware GemFire to accomplish simple tasks, such as building a prototype.

So, Spring Data for VMware GemFire is all about "choice" and Spring Boot for Tanzu GemFire is all about "convention". Together these frameworks provide application developers with convenience and ease to move quickly and reliably.

Conventions

Currently, Spring Boot for Tanzu GemFire provides auto-configuration for the following features:

- `ClientCache`
- Caching with Spring's Cache Abstraction
- Continuous Query
- Function Execution
- Logging
- PDX
- `GemfireTemplate`
- Spring Data Repositories
- Security (auth and SSL)
- Spring Session

This means the following Spring Data for VMware GemFire annotations are not required to use the features above:

- `@ClientCacheApplication`
- `@EnableGemfireCaching` (or by using Spring Framework's `@EnableCaching` annotation)
- `@EnableContinuousQueries`
- `@EnableGemfireFunctionExecutions`
- `@EnableLogging`
- `@EnablePdx`
- `@EnableGemfireRepositories`
- `@EnableSecurity`
- `@EnableSsl`
- `@EnableGemFireHttpSession`

Since Spring Boot for Tanzu GemFire auto-configures these features for you, the above annotations are not strictly required. Typically, you would only declare one of these annotations when you want to "override" Spring Boot's conventions, as 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.

Security

As with the `@ClientCacheApplication` annotation, the `@EnableSecurity` annotation is not strictly required, 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 by using the well-known and documented Spring Data for VMware GemFire username and password properties in Spring Boot `application.properties`:

Example 2. Required Security Properties in a Non-Manage Environment

```
spring.data.gemfire.security.username=MyUser
spring.data.gemfire.security.password=Secret
```

You need not explicitly declare the `@EnableSecurity` annotation just to specify security configuration (such as username and password).

Inside a managed environment, such as the Tanzu Platform for Cloud Foundry when using Tanzu GemFire, Spring Boot for Tanzu GemFire is able to introspect the environment and configure security (auth) completely without the need to specify any configuration, usernames and passwords, or otherwise. This is due, in part, because Tanzu Platform for Cloud Foundry supplies the security details in the VCAP environment when the application is deployed to the platform and bound to services (such as GemFire).

So, in short, you need not explicitly declare the `@EnableSecurity` annotation (or `@ClientCacheApplication`).

However, if you do explicitly declare the `@ClientCacheApplication` or `@EnableSecurity` annotations, you are now responsible for this configuration, and Spring Boot for Tanzu GemFire's auto-configuration no longer applies.

While explicitly declaring `@EnableSecurity` makes more sense when "overriding" the Spring Boot for Tanzu GemFire 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 VMware GemFire, because, in certain cases (such as security), not even Spring is able to completely shield you from the nuances of VMware GemFire's configuration. No framework can.

You must configure both auth and SSL before the cache instance is created. This is because security is enabled and configured during the "construction" of the cache. Also, 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 Spring Boot for Tanzu GemFire 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 Spring Boot for Tanzu GemFire must be triggered before the `ClientCache` auto-configuration class, which is why a `ClientCache` instance cannot "auto" authenticate properly in GemFire for Tanzu Platform for Cloud Foundry when the `@ClientCacheApplication` is explicitly declared without some assistance. In other words you must also explicitly declare the `@EnableSecurity` annotation in this case, since you overrode the auto-configuration of the cache, and implicitly security, as well.

Again, this is due to the way security (auth) and SSL metadata must be supplied to GemFire on startup.

Extension

Most of the time, many of the other auto-configured annotations for CQ, Functions, PDX, Repositories, and so on need not ever be declared explicitly.

Many of these features are enabled automatically by having Spring Boot for Tanzu GemFire or other libraries (such as Spring Session) on the application classpath or are enabled based on other annotations applied to beans in the Spring `ApplicationContext`.

We review a few examples in the following sections.

Caching

It is rarely, if ever, necessary to explicitly declare either the Spring Framework's `@EnableCaching` or the Spring Data for VMware GemFire-specific `@EnableGemfireCaching` annotation in Spring configuration when you use Spring Boot for Tanzu GemFire. Spring Boot for Tanzu GemFire automatically enables caching and configures the Spring Data for VMware GemFire `GemfireCacheManager` for you.

You need only focus on which application service components are appropriate for caching:

Example 3. Service Caching

```
@Service
class CustomerService {

    @Autowired
    private CustomerRepository customerRepository;

    @Cacheable("CustomersByName")
    public Customer findBy(String name) {
        return customerRepository.findByName(name);
    }
}
```

You need to create VMware GemFire Regions that back the caches declared in your application service components (`CustomersByName` in the preceding example) by using Spring's caching annotations (such as `@Cacheable`), or alternatively, JSR-107 JCache annotations (such as `@CacheResult`).

You can do that by defining each Region explicitly or, more conveniently, you can use the following approach:

Example 4. Configuring Caches (Regions)

```
@SpringBootApplication
@EnableCachingDefinedRegions
class Application {  }
```

`@EnableCachingDefinedRegions` is optional, provided for convenience, and complementary to caching when used rather than being necessary.

Continuous Query

It is rarely, if ever, necessary to explicitly declare the Spring Data for VMware GemFire `@EnableContinuousQueries` annotation. Instead, you should focus on defining your application queries and worry less about the plumbing.

Consider the following example:

Example 5. Defining Queries for CQ

```
@Component
public class TemperatureMonitor extends AbstractTemperatureEventPublisher {

    @ContinuousQuery(name = "BoilingTemperatureMonitor",
        query = "SELECT * FROM /TemperatureReadings WHERE temperature.measurement >= 2
12.0")
```

```
    public void boilingTemperatureReadings(CqEvent event) {
        publish(event, temperatureReading -> new BoilingTemperatureEvent(this, tempera
tureReading));
    }

    @ContinuousQuery(name = "FreezingTemperatureMonitor",
        query = "SELECT * FROM /TemperatureReadings WHERE temperature.measurement <= 3
2.0")
    public void freezingTemperatureReadings(CqEvent event) {
        publish(event, temperatureReading -> new FreezingTemperatureEvent(this, temper
atureReading));
    }
}
```

VMware GemFire CQ applies only to clients.

Functions

You rarely, if ever, need to explicitly declare either the `@EnableGemfireFunctionExecutions`annotation since Spring Boot for Tanzu GemFire provides auto-configuration for Function executions.

Example 6. Function Execution

You must first define the function and deploy it to the server, then define the following interface.

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

    Object exampleFunction(Object arg);

}
```

Spring Boot for Tanzu GemFire automatically creates execution 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 executions (interfaces) should exist below the `@SpringBootApplication` annotated main class.

PDX

You rarely, if ever, need to explicitly declare the `@EnablePdx` annotation, since Spring Boot for Tanzu GemFire auto-configures PDX by default. Spring Boot for Tanzu GemFire also automatically configures the Spring Data for VMware GemFire `MappingPdxSerializer` as the default `PdxSerializer`.

It is easy to customize the PDX configuration by setting the appropriate properties (search for “PDX”) in Spring Boot `application.properties`.

Spring Data Repositories

You rarely, if ever, need to explicitly declare the `@EnableGemfireRepositories` annotation, since Spring Boot for Tanzu GemFire auto-configures Spring Data (SD) Repositories by default.

You need only define your Repositories:

Example 7. Customer’s Repository

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

    Customer findByName(String name);

}
```

Spring Boot for Tanzu GemFire finds the Repository interfaces defined in your application, proxies them, and registers them as beans in the Spring `ApplicationContext`. The Repositories can be injected into other application service components.

It is sometimes convenient to use the `@EnableEntityDefinedRegions` along with Spring Data Repositories to identify the entities used by your application and define the Regions used by the Spring Data Repository infrastructure to persist the entity's state. The `@EnableEntityDefinedRegions` annotation is optional, provided for convenience, and complementary to the `@EnableGemfireRepositories` annotation.

Explicit Configuration

Most of the other annotations provided in Spring Data for VMware GemFire are focused on particular application concerns or enable certain VMware GemFire features, rather than being a necessity, including:

- `@EnableAutoRegionLookup`
- `@EnableBeanFactoryLocator`
- `@EnableCachingDefinedRegions`
- `@EnableClusterDefinedRegions`
- `@EnableCompression`
- `@EnableDiskStore(s)`
- `@EnableEntityDefinedRegions`
- `@EnableEviction`
- `@EnableExpiration`
- `@EnableGemFireAsLastResource`
- `@EnablePool(s)`
- `@EnableStatistics`
- `@UseGemFireProperties`

None of these annotations are necessary and none are auto-configured by Spring Boot for Tanzu GemFire. They are at your disposal when and if you need them. This also means that none of these annotations are in conflict with any Spring Boot for Tanzu GemFire auto-configuration.

Summary

In conclusion, you need to understand where Spring Data for VMware GemFire ends and Spring Boot for Tanzu GemFire begins. It all begins with the auto-configuration provided by Spring Boot for Tanzu GemFire.

If a feature or function is not covered by Spring Boot for Tanzu GemFire's auto-configuration, you are responsible for enabling and configuring the feature appropriately, as needed by your application.

In other cases, you might also want to explicitly declare a complimentary annotation (such as `@EnableEntityDefinedRegions`) for convenience, since Spring Boot for Tanzu GemFire provides no convention or opinion.

In all remaining cases, it boils down to understanding how VMware GemFire works under the hood. While we go to great lengths to shield you from as many details as possible, it is not feasible or practical to address all matters, such as cache creation and security.

Configuration Metadata Reference

This topic lists properties recognized and processed by Spring Data for VMware Tanzu GemFire and Spring Session for VMware Tanzu GemFire.

You can use these properties in Spring Boot `application.properties` or as JVM System properties, to configure different aspects of or enable individual features of VMware GemFire in a Spring application. When combined with the power of Spring Boot, they allow you to quickly create an application that uses VMware GemFire.

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`.

Miscellaneous Properties

Name	Description	Default	From
`name`	Name of the VMware GemFire application.	`SpringBasedCacheClientApplication`	`ClientCacheApplication.name`
`use-bean-factory-locator`	Enable the Spring Data for VMware GemFire `BeanFactoryLocator` when mixing Spring config with VMware GemFire native config (such as `cache.xml`) and you wish to configure VMware GemFire objects declared in `cache.xml` with Spring.	`false`	`ClientCacheApplication.useBeanFactoryLocator`

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`

Name	Description	Default	From
`cache.critical-heap-percentage`	Percentage of heap at or above which the cache is considered in danger of becoming inoperable.		`ClientCacheApplication.criticalHeapPercentage`
`cache.critical-off-heap-percentage`	Percentage of off-heap at or above which the cache is considered in danger of becoming inoperable.		`ClientCacheApplication.criticalOffHeapPercentage`
`cache.enable-auto-region-lookup`	Whether to lookup Regions configured in VMware GemFire native configuration and declare them as Spring beans.	`false`	`EnableAutoRegionLookup.enable`
`cache.eviction-heap-percentage`	Percentage of heap at or above which the eviction should begin on Regions configured for HeapLRU eviction.		`ClientCacheApplication.evictionHeapPercentage`
`cache.eviction-off-heap-percentage`	Percentage of off-heap at or above which the eviction should begin on Regions configured for HeapLRU eviction.		`ClientCacheApplication.evictionOffHeapPercentage`
`cache.log-level`	Configure the log-level of a VMware 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 that implements `org.apache.geode.compression.Compressor`.		`EnableCompression.compressorBeanName`
`cache.compression.region-names`	Comma-delimited list of Region names for which compression is configured.	`[]`	`EnableCompression.RegionNames`

ClientCache Properties

Table 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 used by servers to reestablish any messaging that was interrupted by client downtime.		`ClientCacheApplication.durableClientId`
`cache.client.durable-client-timeout`	Used only for clients in a client/server installation. Number of seconds this client can remain disconnected from its server and have the server continue to accumulate durable events for it.	`300`	`ClientCacheApplication.durableClientTimeout`
`cache.client.keep-alive`	Whether the server should keep the durable client's queues alive for the timeout period.	`false`	`ClientCacheApplication.keepAlive`

DiskStore Properties

Name	Description	Default	From
`disk.store.allow-force-compaction`	Whether to allow `DiskStore.forceCompaction()` to be called on Regions that use a disk store.	`false`	`EnableDiskStore.allowForceCompaction`
`disk.store.auto-compact`	Whether to cause the disk files to be automatically compacted.	`true`	`EnableDiskStore.autoCompact`
`disk.store.compaction-threshold`	The threshold at which an oplog becomes compactible.	`50`	`EnableDiskStore.compactionThreshold`
`disk.store.directory.location`	The system directory where the `DiskStore` (oplog) files are stored.	`[]`	`EnableDiskStore.diskDirectories.location`

Name	Description	Default	From
`disk.store.directory.size`	The amount of disk space allowed to store disk store (oplog) files.	`21474883647`	`EnableDiskStore.diskDirectories.size`
`disk.store.disk-usage-critical-percentage`	The critical threshold for disk usage as a percentage of the total disk volume.	`99.0`	`EnableDiskStore.diskUsageCriticalPercentage`
`disk.store.disk-usage-warning-percentage`	The warning threshold for disk usage as a percentage of the total disk volume.	`90.0`	`EnableDiskStore.diskUsageWarningPercentage`
`disk.store.max-oplog-size`	The maximum size (in megabytes) a single oplog (operation log) can be.	`1024`	`EnableDiskStore.maxOplogSize`
`disk.store.queue-size`	The maximum number of operations that can be asynchronously queued.		`EnableDiskStore.queueSize`
`disk.store.time-interval`	The number of milliseconds that can elapse before data written asynchronously is flushed to disk.	`1000`	`EnableDiskStore.timeInterval`
`disk.store.segments`	The number of segments the DiskStore will be organized into.	`0`	`EnableDiskStore.segments`
`disk.store.write-buffer-size`	Configures the write buffer size in bytes.	`32768`	`EnableDiskStore.writeBufferSize`

`DiskStore` properties can be further targeted at specific `DiskStore` instances by setting the `DiskStore.name` property (see VMware GemFire Java API Reference).

For example, you can specify directory location of the files for a specific, named `DiskStore` by using:

```
spring.data.gemfire.disk.store.Example.directory.location=/path/to/gemfire/disk-stores/Example/
```

The directory location and size of the `DiskStore` files can be further divided into multiple locations and size using array syntax:

```
spring.data.gemfire.disk.store.Example.directory[0].location=/path/to/gemfire/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/gemfire/disk-sto
res/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 `DiskStore` instances. Without array indexes, all named `DiskStore` files are stored in the specified location and limited to the defined size.

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`

Logging Properties

Name	Description	Default	From
`logging.level`	The log level of an VMware GemFire cache. Alias for 'spring.data.gemfire.cache.log-level'.	`config`	`EnableLogging.logLevel`
`logging.log-disk-space-limit`	The amount of disk space allowed to store log files.		`EnableLogging.logDiskSpaceLimit`
`logging.log-file`	The pathname of the log file used to log messages.		`EnableLogging.logFile`
`logging.log-file-size`	The maximum size of a log file before the log file is rolled.		`EnableLogging.logFileSize`

PDX Properties

Name	Description	Default	From
`pdx.disk-store-name`	The name of the `DiskStore` used to store PDX type metadata to disk when PDX is persistent.		`EnablePdx.diskStoreName`

Name	Description	Default	From
`pdx.ignore-unread-fields`	Whether PDX ignores fields that were unread during deserialization.	`false`	`EnablePdx.ignoreUnreadFields`
`pdx.persistent`	Whether PDX persists type metadata to disk.	`false`	`EnablePdx.persistent`
`pdx.read-serialized`	Whether a Region entry is returned as a `PdxInstance` or deserialized back into object form on read.	`false`	`EnablePdx.readSerialized`
`pdx.serialize-bean-name`	The name of a custom Spring bean that implements `org.apache.geode.pdx.PdxSerializer`.		`EnablePdx.serializerBeanName`

Pool Properties

Name	Description	Default	From
`pool.free-connection-timeout`	The timeout used to acquire a free connection from a Pool.	`10000`	`EnablePool.freeConnectionTimeout`
`pool.idle-timeout`	The amount of time a connection can be idle before expiring (and closing) the connection.	`5000`	`EnablePool.idleTimeout`
`pool.load-conditioning-interval`	The interval for how frequently the Pool checks 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 of `locator1[port1],..,locatorN[portN]`		`EnablePool.locators`<

Name	Description	Default	From
`pool.max-connections`	The maximum number of client to server connections that a Pool will create.		`EnablePool.maxConnections`
`pool.min-connections`	The minimum number of client to server connections that a Pool maintains.	`1`	`EnablePool.minConnections`
`pool.multi-user-authentication`	Whether the created Pool can be used by multiple authenticated users.	`false`	`EnablePool.multiUserAuthentication`
`pool.ping-interval`	How often to ping servers to verify that they are still alive.	`10000`	`EnablePool.pingInterval`
`pool.pr-single-hop-enabled`	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 that host Regions with `DataPolicy.PARTITION`.	`true`	`EnablePool.prSingleHopEnabled`
`pool.read-timeout`	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`	Whether to signal the server that the client is prepared and ready to receive events.	`false`	`ClientCacheApplication.readyForEvents`
`pool.retry-attempts`	The number of times to retry a request after timeout/exception.		`EnablePool.retryAttempts`
`pool.server-group`	The group that all servers to which a Pool connects must belong.		`EnablePool.serverGroup`

Name	Description	Default	From
`pool.servers`	Comma-delimited list of `CacheServer` endpoints in the format of `server1[port1],...,serverN[portN]`		`EnablePool.servers`
`pool.socket-buffer-size`	The socket buffer size for each connection made in all Pools.	`32768`	`EnablePool.socketBufferSize`
`pool.statistic-interval`	How often to send client statistics to the server.		`EnablePool.statisticInterval`
pool.subscription-ack-interval	The interval in milliseconds to wait before sending acknowledgements to the `CacheServer` for events received from the server subscriptions.	`100`	`EnablePool.subscriptionAckInterval`
`pool.subscription-enabled`	Whether the created Pool has server-to-client subscriptions enabled.	`false`	`EnablePool.subscriptionEnabled`
`pool.subscription-message-tracking-timeout`	The `messageTrackingTimeout` attribute, which is the time-to-live period, in milliseconds, for subscription events the client has received from the server.	`900000`	`EnablePool.subscriptionMessageTrackingTimeout`
`pool.subscription-redundancy`	The redundancy level for all Pools server-to-client subscriptions.		`EnablePool.subsriptionRedundancy`
`pool.thread-local-connections`	The thread local connections policy for all Pools.	`false`	`EnablePool.threadLocalConnections`

Security Properties

Name	Description	Default	From
`security.username`	The name of the user used to authenticate with the servers.		`EnableSecurity.securityUsername`
`security.password`	The user password used to authenticate with the servers.		`EnableSecurity.securityPassword`
`security.client.authentication-initializer`	Static creation method that returns an `AuthInitialize` object, which obtains credentials for peers in a cluster.		`EnableSecurity.clientAuthentiationInitializer`
`security.manager.class-name`	The name of a class that implements `org.apache.geode.security.SecurityManager`.		`EnableSecurity.securityManagerClassName`
`security.peer.authentication-initializer`	Static creation method that returns an `AuthInitialize` object, which obtains credentials for peers in a cluster.		`EnableSecurity.peerAuthenticationInitializer`
`security.post-processor.class-name`	The name of a class that implements the `org.apache.geode.security.PostProcessor` interface that can be used to change the returned results of Region get operations.		`EnableSecurity.securityPostProcessorClassName`
`security.shiro.ini-resource-path`	The VMware GemFire System property that refers to the location of an Apache Shiro INI file that configures the Apache Shiro Security Framework in order to secure VMware GemFire.		`EnableSecurity.shiroIniResourcePath`

SSL Properties

Name	Description	Default	From
`security.ssl.certificate.alias.cluster`	The alias to the stored SSL certificate used by the cluster to secure communications.		`EnableSsl.componentCertificateAliases`
`security.ssl.certificate.alias.default-alias`	The default alias to the stored SSL certificate used to secure communications across the entire VMware GemFire system.		`EnableSsl.defaultCertificateAlias`
`security.ssl.certificate.alias.gateway`	The alias to the stored SSL certificate used by the WAN Gateway Senders/Receivers to secure communications.		`EnableSsl.componentCertificateAliases`
`security.ssl.certificate.alias.jmx`	The alias to the stored SSL certificate used by the Manager's JMX-based JVM MBeanServer and JMX clients to secure communications.		`EnableSsl.componentCertificateAliases`
`security.ssl.certificate.alias.locator`	The alias to the stored SSL certificate used by the Locator to secure communications.		`EnableSsl.componentCertificateAliases`
`security.ssl.certificate.alias.server`	The alias to the stored SSL certificate used by clients and servers to secure communications.		`EnableSsl.componentCertificateAliases`
`security.ssl.certificate.alias.web`	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`

Name	Description	Default	From
`security.ssl.components`	Comma-delimited list of VMware GemFire components (for example, WAN) to be configured for SSL communication.		`EnableSsl.components`
`security.ssl.keystore`	The system pathname to the Java KeyStore file storing certificates for SSL.		`EnableSsl.keystore`
`security.ssl.keystore.password`	The password used to access the Java KeyStore file.		`EnableSsl.keystorePassword`
`security.ssl.keystore.type`	The password used to access the Java KeyStore file (for example, JKS).		`EnableSsl.keystoreType`
`security.ssl.protocols`	Comma-separated list of SSL protocols or `any`.		`EnableSsl.protocols`
`security.ssl.require-authentication`	Whether two-way authentication is required.		`EnableSsl.requireAuthentication`
`security.ssl.truststore`	The system pathname to the trust store (Java KeyStore file) that stores certificates for SSL.		`EnableSsl.truststore`
`security.ssl.truststore.password`	The password used to access the trust store (Java KeyStore file).		`EnableSsl.truststorePassword`
`security.ssl.truststore.type`	The password used to access the trust store (ex. Java KeyStore (JKS) file).		`EnableSsl.truststoreType`
`security.ssl.web-require-authentication`	Whether two-way HTTP authentication is required.	`false`	`EnableSsl.webRequireAuthentication`

Service Properties

Name	Description	Default	From
service.http.bind-address	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	The port used by the embedded HTTP server to listen for HTTP(S) connections.	7070	EnableHttpService.port
service.http.ssl-require-authentication	Whether two-way HTTP authentication is required.	false	EnableHttpService.sslRequireAuthentication
service.http.dev-rest-api-start	Whether to start the Developer REST API web service. A full installation of VMware GemFire is required, and you must set the $GEMFIRE_HOME environment variable.	false	EnableHttpService.startDeveloperRestApi

Spring Session-Based Properties

The following properties all have a spring.session.data.gemfire.* prefix. For example, to set the session Region name, set spring.session.data.gemfire.session.region.name in Spring Boot application.properties.

Spring Session Properties

Name	Description	Default	From
cache.client.pool.name	Name of the pool used to send data access operations between the client and servers.	gemfirePool	EnableGemFireHttpSession.poolName
cache.client.Region.shortcut	The DataPolicy used by the client Region to manage (HTTP) session state.	ClientRegionShortcut.PROXY	EnableGemFireHttpSession.clientRegionShortcut
session.attributes.indexable	The names of session attributes for which an Index is created.	[]	EnableGemFireHttpSession.indexableSessionAttributes

Name	Description	Default	From
`session.expiration.max-inactive-interval-seconds`	Configures the number of seconds in which a session can remain inactive before it expires.	`1800`	`EnableGemFireHttpSession.maxInactiveIntervalSeconds`
`session.Region.name`	The name of the (client/server) Region used to manage (HTTP) session state.	`ClusteredSpringSessions`	`EnableGemFireHttpSession.RegionName`
`session.serializer.bean-name`	The name of a Spring bean that implements `org.springframework.session.data.gemfire.serialization.SessionSerializer`.		`EnableGemFireHttpSession.sessionSerializerBeanName`

While we do not recommend using VMware GemFire properties directly in your Spring applications, Spring Boot for Tanzu GemFire does not prevent you from doing so. See the complete reference to the VMware GemFire specific properties.

VMware GemFire is very strict about the properties that may be specified in a `gemfire.properties` file. You cannot mix Spring properties with `gemfire.*` properties in a VMware GemFire `gemfire.properties` file.

Deactivating Auto-configuration

This topic explains how to deactivate Spring Boot for VMware Tanzu GemFire auto-configuration.

To deactivate the auto-configuration of any feature provided by Spring Boot for Tanzu GemFire, you can specify the auto-configuration class in the `exclude` attribute of the `@SpringBootApplication` annotation:

Example 1. Deactivate Auto-configuration of PDX

```
@SpringBootApplication(exclude = PdxSerializationAutoConfiguration.class)
public class MySpringBootApplication {

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

You can deactivate more than one auto-configuration class at a time by specifying each class in the `exclude` attribute using array syntax:

Example 2. Deactivate Auto-configuration of PDX and 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 Tanzu GemFire includes:

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