Spring Gemfire Integration Reference Guide

1.0.0.M1

Costin Leau (SpringSource, a division of VMware)

Preface

Spring GemFire Integration focuses on integrating Spring Framework's powerful, non-invasive programming model and concepts with Gemstone's GemFire Enterprise Fabric, providing easier configuration, use and high-level abstractions. This document assumes the reader is already has a basic familiarity with the Spring Framework and GemFire concepts and APIs.

While every effort has been made to ensure that this documentation is comprehensive and there are no errors, nevertheless some topics might require more explanation and some typos might have crept in. If you do spot any mistakes or even more serious errors and you can spare a few cycles during lunch, please do bring the error to the attention of the Spring GemFire Integration team by raising an issue. Thank you.

Part I. Introduction

This document is the reference guide for Spring GemFire project (SGF). It explains the relationship between Spring framework and GemFire Enterprise Fabric (GEF) 6.0.x, defines the basic concepts and semantics of the integration and how these can be used effectively.

Part II. Reference Documentation

Document structure

This part of the reference documentation explains the core functionality offered by Spring GemFire integration.

Chapter 1, *Bootstrapping GemFire through the Spring container* describes the configuration support provided for bootstrapping, initializing and accessing a GemFire cache or region.

Chapter 3, *Working with GemFire Serialization* describes the enhancements for GemFire (de)serialization process and management of associated objects.

Chapter 2, *Working with the GemFire APIs* explains the integration between GemFire API and the various "data" features available in Spring, such as transaction management and exception translation.

Chapter 4, *Sample Applications* describes the samples provided with the distribution for showcasing the various features available in Spring GemFire.

Chapter 1. Bootstrapping GemFire through the Spring container

One of the first tasks when using GemFire and Spring is to configure the data grid through the IoC container. While this is possible out of the box, the configuration tends to be verbose and only address basic cases. To address this problem, the Spring GemFire project provides several classes that enable the configuration of distributed caches or regions to support a variety of scenarios with minimal effort.

1.1. Configuring the GemFire `Cache`

In order to use the GemFire Fabric, one needs to either create a new `Cache` or connect to an existing one. As in the current version of GemFire, there can be only one opened cache per VM (or classloader to be technically correct). In most cases the cache is created once and then all other consumers connect to it.

In its simplest form, a cache can be defined in one line:

```
<bean id="default-cache" class="org.springframework.data.gemfire.CacheFactoryBean"/>
```

Here, the *default-cache* will try to connect to an existing cache and, in case one does not exist, create it. Since no additional properties were specified the created cache uses the default cache configuration.

Especially in environments with opened caches, this basic configuration can go a long way. For scenarios where the cache needs to be configured, the user can pass in a reference the GemFire configuration file:

```
<bean id="cache-with-xml" class="org.springframework.data.gemfire.CacheFactoryBean">
    <property name="cacheXml" value="classpath:cache.xml"/>
</bean>
```

In this example, if the cache needs to be created, it will use the file named `cache.xml` located in the classpath root. Only if the cache is created will the configuration file be used.

Note

Note that the configuration makes use of Spring's `Resource` abstraction to locate the file. This allows various search patterns to be used, depending on the running environment or the prefix specified (if any) by the value.

In addition to referencing an external configuration file one can specify GemFire settings directly through Java `Properties`. This can be quite handy when just a few settings need to be changed:

```
<bean id="cache-with-props" class="org.springframework.data.gemfire.CacheFactoryBean">
    <property name="properties">
        <props>
           <prop key="bind-address">127.0.0.1</prop>
        </props>
    </property>
</bean>
```

So far our examples relied on the primary Spring namespace (`beans`). However one is free to add other namespaces to simplify or enhance the configuration. Let's do the same thing to the configuration above by using the `util` namespace and externalize the properties from the configuration which is a best practice.

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

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:util="http://www.springframework.org/schema/util"
  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/sp
    http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util.xsd">

  <bean id="cache-with-props" class="org.springframework.data.gemfire.CacheFactoryBean">
    <property name="properties">
      <util:properties location="classpath:/deployment/env.properties"/>
    </property>
  </bean>
</beans>
```

It is worth pointing out again, that the cache settings apply only if the cache needs to be created, there is no opened cache in existence otherwise the existing cache will be used and the configuration will simply be discarded.

1.2. Configuring a GemFire `Region`

Once the `Cache` is configured, one needs to configure one or more `Regions` to interact with the data fabric. In a similar manner to the `CacheFactoryBean`, the `RegionFactoryBean` allows existing `Regions` to retrieved or, in case they don't exist, created using various settings. One can specify the `Region` name, whether it will be destroyed on shutdown (thereby acting as a temporary cache), the associated `CacheLoaders`, `CacheListeners` and `CacheWriters` and if needed, the `RegionAttributes` for full customization.

Let us start with a simple region declaration, named *basic* using a nested cache declaration:

```
<bean id="basic" class="org.springframework.data.gemfire.RegionFactoryBean">
  <property name="cache">
      <bean class="org.springframework.data.gemfire.CacheFactoryBean"/>
  </property>
  <property name="name" value="basic"/>
</bean>
```

Since the region bean definition name is usually the same with that of the cache, the `name` property can be omitted (the bean name will be used automatically). Additionally by using the name the `p` namespace, the configuration can be simplified even more:

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

    <!-- shared cache across regions -->
    <bean id="cache" class="org.springframework.data.gemfire.CacheFactoryBean"/>

    <!-- region named 'basic' -->
    <bean id="basic" class="org.springframework.data.gemfire.RegionFactoryBean" p:cache-ref="cache"/>

    <!-- region with a name different then the bean definition -->
    <bean id="root-region" class="org.springframework.data.gemfire.RegionFactoryBean" p:cache-ref="cache" p:name
</beans>
```

It is worth pointing out, that for the vast majority of cases configuring the cache loader, listener and writer through the Spring container is preferred since the same instances can be reused across multiple regions and additionally, the instances themselves can benefit from the container's rich feature set:

```
<bean id="cacheLogger" class="org.some.pkg.CacheLogger"/>
<bean id="customized-region" class="org.springframework.data.gemfire.RegionFactoryBean" p:cache-ref="cache">
  <property name="cacheListeners">
    <array>
      <ref name="cacheLogger"/>
```

```
    <bean class="org.some.other.pkg.SysoutLogger"/>
    </array>
  </property>
  <property name="cacheLoader"><bean class="org.some.pkg.CacheLoad"/></property>
  <property name="cacheWriter"><bean class="org.some.pkg.CacheWrite"/></property>
</bean>

<bean id="local-region" class="org.springframework.data.gemfire.RegionFactoryBean" p:cache-ref="cache">
  <property name="cacheListeners" ref="cacheLogger"/>
</bean>
```

1.2.1. Configuring a *client* `Region`

For scenarios where a *CacheServer* is used and *clients* need to be configured, SGI offers a dedicated configuration class named: `ClientRegionFactoryBean`. This allows client *interests* to be registered in both key and regex form through `Interest` and `RegexInterest` classes in the `org.springframework.data.gemfire` package:

```
<bean id="interested-client" class="org.springframework.data.gemfire.ClientRegionFactoryBean" p:cache-ref="cache
  <property name="interests">
    <array>
      <!-- key-based interest -->
      <bean class="org.springframework.data.gemfire.Interest" p:key="Vlaicu" p:policy="NONE"/>
      <!-- regex-based interest -->
      <bean class="org.springframework.data.gemfire.RegexInterest" p:key=".*" p:policy="KEYS" p:durable="true"/>
    </array>
  </property>
</bean>
```

1.2.2. Advanced configuration through a `Region`'s *attributes*

Users that need fine control over a region, can configure it in Spring by using the `attributes` property. To ease declarative configuration in Spring, SGI provides two `FactoryBeans` for creating `RegionAttributes` and `PartitionAttributes`, namely `RegionAttributesFactory` and `PartitionAttributesFactory`. See below an example of configuring a partitioned region through Spring XML:

```
<bean id="partitioned-region" class="org.springframework.data.gemfire.RegionFactoryBean" p:cache-ref="cache">
  <property name="attributes">
    <bean class="org.springframework.data.gemfire.RegionAttributesFactory" p:initial-capacity="1024">
      <property name="partitionAttributes">
        <bean class="org.springframework.data.gemfire.PartitionAttributesFactory" p:redundant-copies="2" p:local
      </property>
    </bean>
  </property>
</bean>
```

By using the attribute factories above, one can reduce the size of the `cache.xml` or even eliminate it all together.

Chapter 2. Working with the GemFire APIs

Once the GemFire cache and regions have been configured they can injected and used inside application objects. This chapter describes the integration with Spring's transaction management functionality and `DaoException` hierarchy. It also covers support for dependency injection of GemFire managed objects.

2.1. Exception translation

Using a new data access technology requires not just accommodating to a new API but also handling exceptions specific to that technology. To accommodate this case, Spring Framework provides a technology agnostic, consistent exception hierarchy that abstracts one from proprietary (and usually checked) exceptions to a set of focused runtime exceptions. As mentioned in the Spring Framework documentation, exception translation can be applied transparently to your data access objects through the use of the `@Repository` annotation and AOP by defining a PersistenceExceptionTranslationPostProcessor bean. The same exception translation functionality is enabled when using Gemfire as long as at least a `CacheFactoryBean` is declared. The `Cache` factory acts as an exception translator which is automatically detected by the Spring infrastructure and used accordingly.

2.2. `GemfireTemplate`

As with many other high-level abstractions provided by the Spring Framework and related projects, Spring GemFire provides a *template* that plays a central role when working with the GemFire API. The class provides several *one-liner* methods, for popular operations but also the ability to *execute* code against the native GemFire API without having to deal with exceptions for example through the `GemfireCallback`.

The template class requires a GemFire `Region` instance and once configured is thread-safe and should be reused across multiple classes:

```
<bean id="gemfireTemplate" class="org.springframework.data.gemfire.GemfireTemplate" p:region-ref="someRegion"/>
```

Once the template is configured, one can use it alongside `GemfireCallback` to work directly with the GemFire `Region`, without having to deal with checked exceptions, threading or resource management concerns:

```
template.execute(new GemfireCallback<Iterable<String>>() {
  public Iterable<String> doInGemfire(Region reg) throws GemFireCheckedException, GemFireException {
    // working against a Region of String
    Region<String, String> region = reg;

    region.put("1", "one");
    region.put("3", "three");

    // should return 1
    return region.query("length < 5).size();
  }
});
```

2.3. Transaction Management

One of the most popular features of Spring Framework is transaction management. If you are not familiar with it, we strongly recommend looking into it as it offers a consistent programming model that works transparently across multiple APIs that can be configured either programmatically or declaratively (the most popular choice).

For Gemfire, SGI provides a dedicated, per-cache, transaction manager that once declared, allows actions on the `Regions` to be grouped and executed atomically through Spring:

```
<bean id="transaction-manager" class="org.springframework.data.gemfire.GemfireTransactionManager" p:cache-ref="c
```

Note that currently GemFire supports optimistic transactions with *read committed* isolation. Furthermore, to guarantee this isolation, developers should avoid making *in-place* changes, that is manually modifying the values present in the cache. To prevent this from happening, the transaction manager configured the cache to use *copy on read* semantics, meaning a clone of the actual value is created, each time a read is performed. This behaviour can be disabled if needed through the `copyOnRead` property. For more information on the semantics of the underlying GemFire transaction manager, see the GemFire documentation.

2.4. Wiring `Declarable` components

GemFire XML configuration (usually named `cache.xml` allows *user* objects to be declared as part of the fabric configuration. Usually these objects are `CacheLoaders` or other pluggable components into GemFire. Out of the box in GemFire, each such type declared through XML must implement the `Declarable` interface which allows arbitrary parameters to be passed to the declared class through a `Properties` instance.

In this section we describe how you can configure the pluggable components defined in `cache.xml` using Spring while keeping your Cache/Region configuration defined in `cache.xml` This allows your pluggable components to focus on the application logic and not the location or creation of DataSources or other collaboration object.

However, if you are starting on a green-field project, it is recommended that you configure Cache, Region, and other pluggable components directly in Spring. This avoids inheriting from the `Declarable` interface or the base class presented in this section. See the following sidebar for more information on this approach.

> **Eliminate `Declarable` components**
>
> One can configure custom types entirely inside through Spring as mentioned in Section 1.2, “Configuring a GemFire `Region`”. That way, one does not have to implement the `Declarable` interface and gets access to all the features of the Spring IoC container (including not just dependency injection but also life-cycle and instance management).

As an example of configuring a `Declarable` component using Spring, consider the following declaration (taken from the `Declarable` javadoc):

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <parameter name="URL">
     <string>jdbc://12.34.56.78/mydb</string>
   </parameter>
</cache-loader>
```

To simplify the task of parsing, converting the parameters and initializing the object, SGI offers a base class (`WiringDeclarableSupport`) that allows GemFire user objects to be wired through a *template* bean definition or, in case that is missing perform autowiring through the Spring container. To take advantage of this feature, the user objects need to extend `WiringDeclarableSupport` which automatically locates the declaring `BeanFactory` and performs wiring as part of the initialization process.

Why is a base class needed?

In the current GemFire release there is no concept of an *object factory* and the types declared are instantiated and used as is - that is there are no other ways in which third parties can take care of the object creation outside GemFire. Support for this feature is planned for the up-coming GemFire release (6.5)

2.4.1. Configuration using *template* definitions

When used `WiringDeclarableSupport` tries to first locate an existing bean definition and use that as wiring template. Unless specified, the component class name will be used as an implicit bean definition name. Let's see how our `DBLoader` declaration would look in that case:

```
public class DBLoader extends WiringDeclarableSupport implements CacheLoader {
  private DataSource dataSource;

  public void setDataSource(DataSource ds){
    this.dataSource = ds;
  }

  public Object load(LoaderHelper helper) { ... }
}
```

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- no parameter is passed (use the bean implicit name
   that is the class name) -->
</cache-loader>
```

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

  <bean id="dataSource" ... />

     <!-- template bean definition -->
     <bean id="com.company.app.DBLoader" abstract="true" p:dataSource-ref="dataSource"/>
</beans>
```

In the scenario above, as no parameter was specified, a bean with id/name `com.company.app.DBLoader` was searched for. The found bean definition is used as a template for wiring the instance created by GemFire. For cases where the bean name uses a different convention, one can pass in the `bean-name` parameter in the GemFire configuration:

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- pass the bean definition template name
        as parameter -->
   <parameter name="bean-name">
     <string>template-bean</string>
   </parameter>
</cache-loader>
```

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

```
          http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="dataSource" ... />

     <!-- template bean definition -->
     <bean id="template-bean" abstract="true" p:dataSource-ref="dataSource"/>

</beans>
```

Note

The *template* bean definitions do not have to be declared in XML - any format is allowed (Groovy, annotations, etc..).

2.4.2. Configuration using auto-wiring and annotations

If no bean definition is found, by default, `WiringDeclarableSupport` will autowire the declaring instance. This means that unless any dependency injection *metadata* is offered by the instance, the container will find the object setters and try to automatically satisfy these dependencies. However, one can also use JDK 5 annotations to provide additional information to the auto-wiring process. We strongly recommend reading the dedicated chapter in the Spring documentation for more information on the supported annotations and enabling factors.

For example, the hypothetical `DBLoader` declaration above can be injected with a Spring-configured DataSource in the following way:

```
public class DBLoader extends WiringDeclarableSupport implements CacheLoader {
  // use annotations to 'mark' the needed dependencies
  @javax.inject.Inject
  private DataSource dataSource;

  public Object load(LoaderHelper helper) { ... }
}
```

```
<cache-loader>
   <class-name>com.company.app.DBLoader</class-name>
   <!-- no need to declare any parameters anymore
        since the class is auto-wired -->
</cache-loader>
```

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

     <!-- enable annotation processing -->
     <context:annotation-config/>

</beans>
```

By using the JSR-330 annotations, the cache loader code has been simplified since the location and creation of the DataSource has been externalized and the user code is concerned only with the loading process. The `DataSource` might be transactional, created lazily, shared between multiple objects or retrieved from JNDI - these aspects can be easily configured and changed through the Spring container without touching the `DBLoader` code.

Chapter 3. Working with GemFire Serialization

To improve overall performance of the data fabric, GemFire supports a dedicated serialization protocol that is both faster and offers more compact results over the standard Java serialization and works transparently across various language platforms (such as Java, .NET and C++). This chapter discusses the various ways in which SGI simplifies and improves GemFire custom serialization in Java.

3.1. Wiring deserialized instances

It is fairly common for serialized objects to have transient data. Transient data is often dependent on the node or environment where it lives at a certain point in time, for example a DataSource. Serializing such information is useless (and potentially even dangerous) since it is local to a certain VM/machine. For such cases, SGI offers a special `Instantiator` that performs wiring for each new instance created by GemFire during deserialization.

Through such a mechanism, one can rely on the Spring container to inject (and manage) certain dependencies making it easy to split transient from persistent data and have *rich domain objects* in a transparent manner (Spring users might find this approach similar to that of `@Configurable`). The `WiringInstantiator` works just like `WiringDeclarableSupport`, trying to first locate a bean definition as a wiring template and following to autowiring otherwise. Please refer to the previous section (Section 2.4, “Wiring `Declarable` components”) for more details on wiring functionality.

To use this `Instantiator`, simply declare it as a usual bean:

```
<bean id="instantiator" class="org.springframework.data.gemfire.serialization.WiringInstantiator">
  <!-- DataSerializable type -->
  <constructor-arg>org.pkg.SomeDataSerializableClass</constructor-arg>
  <!-- type id -->
  <constructor-arg>95</constructor-arg>
</bean>
```

During the container startup, once it is being initialized, the `instantiator` will, by default, register itself with the GemFire system and perform wiring on all instances of `SomeDataSerializableClass` created by GemFire during deserialization.

3.2. Auto-generating custom `Instantiator`s

For data intensive applications, a large number of instances might be created on each machine as data flows in. Out of the box, GemFire uses reflection to create new types but for some scenarios, this might prove to be expensive. As always, it is good to perform profiling to quantify whether this is the case or not. For such cases, SGI allows the automatic generation of `Instatiator` classes which instantiate a new type (using the default constructor) without the use of reflection:

```
<bean id="instantiator-factory" class="org.springframework.data.gemfire.serialization.InstantiatorFactoryBean">
  <property name="customTypes">
    <map>
      <entry key="org.pkg.CustomTypeA" value="1025"/>
      <entry key="org.pkg.CustomTypeB" value="1026"/>
    </map>
  </property>
</bean>
```

The definition above, automatically generated two `Instantiators` for two classes, namely `CustomTypeA` and `CustomTypeB` and registers them with GemFire, under user id `1025` and `1026`. The two instantiators avoid the

use of reflection and create the instances directly through Java code.

Chapter 4. Sample Applications

The Spring GemFire project includes one sample application. Named "Hello World", the sample demonstrates how to configure and use GemFire inside a Spring application. At runtime, the sample offers a *shell* to the user allowing him to run various commands against the grid. It provides an excellent starting point for users unfamiliar with the essential components or the Spring and GemFire concepts.

The sample is bundled with the distribution and is Maven-based. One can easily import them into any Maven-aware IDE (such as SpringSource Tool Suite) or run them from the command-line.

4.1. Hello World

The Hello World sample demonstrates the core functionality of the Spring GemFire project. It bootstraps GemFire, configures it, executes arbitrary commands against it and shuts it down when the application exits. Multiple instances can be started at the same time as they will work with each other sharing data without any user intervention.

4.1.1. Starting and stopping the sample

Hello World is designed as a stand-alone java application. It features a `Main` class which can be started either from your IDE of choice (in Eclipse/STS through `Run As/Java Application`) or from the command line through Maven using `mvn exec:java`. One can also use `java` directly on the resulting artifact if the classpath is properly set.

To stop the sample, simply type `exit` at the command line or press `Ctrl+C` to stop the VM and shutdown the Spring container.

4.1.2. Using the sample

Once started, the sample will create a shared data grid and allow the user to issue commands against it. The output will likely look as follows:

```
INFO: Created GemFire Cache [Spring GemFire World] v. X.Y.Z
INFO: Created new cache region [myWorld]
INFO: Member xxxxxx:50694/51611 connecting to region [myWorld]
Hello World!
Want to interact with the world ? ...
Supported commands are:

get <key> - retrieves an entry (by key) from the grid
put <key> <value> - puts a new entry into the grid
remove <key> - removes an entry (by key) from the grid
...
```

For example to add new items to the grid one can use:

```
-> put 1 unu
INFO: Added [1=unu] to the cache
null
-> put 1 one
INFO: Updated [1] from [unu] to [one]
unu
-> size
1
-> put 2 two
INFO: Added [2=two] to the cache
null
```

```
-> size
2
```

Multiple instances can be created at the same time. Once started, the new VMs automatically see the existing region and its information:

```
INFO: Connected to Distributed System ['Spring GemFire World'=xxxx:56218/49320@yyyyy]
Hello World!
...

-> size
2
-> map
[2=two] [1=one]
-> query length = 3
[one, two]
```

Experiment with the example, start (and stop) as many instances as you want, run various commands in one instance and see how the others react. To preserve data, at least one instance needs to be alive all times - if all instances are shutdown, the grid data is completely destroyed (in this example - to preserve data between runs, see the GemFire documentations).

4.1.3. Hello World Sample Explained

Hello World uses both Spring XML and annotations for its configuration. The initial boostrapping configuration is `app-context.xml` which includes the cache configuration, defined under `cache-context.xml` file and performs classpath scanning for Spring components. The cache configuration defines the GemFire cache, region and for illustrative purposes a simple cache listener that acts as a logger.

The main *beans* are `HelloWorld` and `CommandProcessor` which rely on the `GemfireTemplate` to interact with the distributed fabric. Both classes use annotations to define their dependency and life-cycle callbacks.

Part III. Other Resources

In addition to this reference documentation, there are a number of other resources that may help you learn how to use GemFire and Spring framework. These additional, third-party resources are enumerated in this section.

Chapter 5. Useful Links

- *Spring GemFire Integration Home Page* - here
- *SpringSource blog* - here
- *GemFire Community* - here