1.0.0 M1

Preface iii
I. Introduction 1
II. Reference Documentation 2
    1. Bootstrapping GemFire through the Spring container 3
        1.1. Configuring the GemFire Cache 3
        1.2. Configuring a GemFire Region 4
            1.2.1. Configuring update interest for client Region 5
    2. Working with the GemFire APIs 7
        2.1. Exception translation 7
    3. Sample Applications 8
        3.1. Prerequisites 8
        3.2. Hello World 9
            3.2.1. Compiling, starting and stopping the sample 9
            3.2.2. Using the sample 9
III. Other Resources 11
    4. Useful Links 12

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

Chapter 3, *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 using dependency injection via the Spring 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.

Note

The M2 release of Spring GemFire will include Spring namespaces that will make it even easier to create and configure the cache and specific region types (e.g. replicated, partitioned, etc.).

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 of the current version of GemFire (6.0.x) there can be only one opened cache per application. 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:

```
<object name="default-cache" type="Spring.Data.GemFire.CacheFactoryObject, Spring.Data.GemFire"/>
```

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

The name of the cache will be the name of the Spring objectd definition unless you specify a different name using the NameValueCollection property described next. Note, you can specify the name of the underlying DistributedSystem that will be created with the property `DistributedSystemName`

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:

```
<object name="cache-with-xml" type="Spring.Data.GemFire.CacheFactoryObject, Spring.Data.GemFire">
  <property name="CacheXml" value="cache.xml"/>
</object>
```

In this example, if the cache needs to be created, it will use the file named `cache.xml` located in the runtime directory.

In addition to referencing an external configuration file one can specify GemFire settings directly through .NET name value properties. This can be quite handy when just a few settings need to be changed:

```
<object name="cache-with-props" type="Spring.Data.GemFire.CacheFactoryObject, Spring.Data.GemFire">
  <property name="Properties">
    <name-values>
      <add key="log-level" value="warning"/>
    </name-values>
  </property>
</object>
```

The 'name' property is used to set the name of the Cache object. For a complete list of properties refer to the GemFire reference documentation.

Spring object definitions support property replacement through the use of variable replacement. The following configuration allows you to externalize the properties from the Spring configuration file which is a best practice. The Spring configuration file is usually an embedded assembly resource so as to prevent accidental changes in production. There are seven locations supported out of the box in Spring.NET where you can place your externalized configuration data and can be extended to support your own locations. In the following example the configuration flues would come from a name-value configuration section in App/Web.config

```
<object name="cache-with-props" type="Spring.Data.GemFire.CacheFactoryObject, Spring.Data.GemFire">
  <property name="Properties">
    <name-values>
      <add key="name" value="StockCache"/>
      <add key="log-level" value="${cache.log-level}"/>
    </name-values>
  </property>
</object>

<object type="Spring.Objects.Factory.Config.VariablePlaceholderConfigurer, Spring.Core">
   <property name="VariableSources">
      <list>
         <object type="Spring.Objects.Factory.Config.ConfigSectionVariableSource, Spring.Core">
            <property name="SectionNames" value="CacheConfiguration" />
         </object>
      </list>
   </property>
</object>
```

The CacheConfiguration section in App.config would then look like the following

```
<configuration>
  <configSections>
    <section name="CacheConfiguration" type="System.Configuration.NameValueSectionHandler"/>
  </configSections>

  <CacheConfiguration>
    <add key="cache.log-level" value="warning"/>
  </CacheConfiguration>

</configuration>
```

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 CacheFactoryObject, the RegionFactoryObject 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:

```
<object name="basic" type="Spring.Data.GemFire.RegionFactoryObject, Spring.Data.GemFire">
  <property Name="Cache">
    <object type="Spring.Data.GemFire.CacheFactoryObject, Spring.Data.GemFire"/>
  </property>
</object>
```

By default the region name is the name of the object definition unless explicitly specified using the `Name` property.

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:

```
<object name="baseRegion" abstract="true">
  <property name="Endpoints" value="localhost:40404"/>
  <property name="Cache" ref="Cache"/>   <!-- definition not shown here -->
</object>

<object name="listeners" type="Spring.Data.GemFire.RegionFactoryObject, Spring.Data.GemFire"
        parent="baseRegion">
  <property name="CacheListener">
    <object type="Spring.Data.GemFire.Tests.SimpleCacheListener, Spring.Data.GemFire.Tests">
      <!-- set properties or constructor arguments -->
    </object>
  </property>
  <property name="CacheLoader">
    <object type="Spring.Data.GemFire.Tests.SimpleCacheLoader, Spring.Data.GemFire.Tests"/>
  </property>
  <property name="CacheWriter">
    <object type="Spring.Data.GemFire.Tests.SimpleCacheWriter, Spring.Data.GemFire.Tests"/>
  </property>
</object>
```

1.2.1. Configuring update interest for *client* Region

Client *interests* can be registered in both key and regex form through AllKeysInterest, KeyInterest, and RegexInterest classes in the `Spring.Data.GemFire` namespace. Here is an example of how to configure the AllKeys interest in the region.

```
<object name="baseRegion" abstract="true">
  <property name="Endpoints" value="localhost:40404"/>
  <property name="Cache" ref="Cache"/>   <!-- definition not shown here -->
</object>

<object name="basic-interest" type="Spring.Data.GemFire.RegionFactoryObject, Spring.Data.GemFire"
        parent="baseRegion">
  <property name="ClientNotification" value="true"/>
  <property name="interests">
    <list>
      <object type="Spring.Data.GemFire.AllKeysInterest"/>
    </list>
  </property>
</object>
```

To register interest for a set of key, use the KeyInterest class, as shown below from the sample application

```
<object name="Region" type="Spring.Data.GemFire.RegionFactoryObject, Spring.Data.GemFire">
  <property name="Endpoints" value="localhost:40404"/>
  <property name="Cache" ref="Cache"/>
  <property name="Name" value="exampleregion"/>
  <property name="ClientNotification" value="true"/>
  <property name="Interests">
    <list>
      <object type="Spring.Data.GemFire.KeyInterest">
        <property name="Keys">
          <list>
            <object type="GemStone.GemFire.Cache.CacheableString" factory-method="Create">
              <constructor-arg value="Key-123"/>
            </object>
          </list>
        </property>
      </object>
    </list>
  </property>
</object>
```

To register interest based on a regular expression, use the following configuration

```
<object name="Region" type="Spring.Data.GemFire.RegionFactoryObject, Spring.Data.GemFire">
  <property name="Endpoints" value="localhost:40404"/>
  <property name="Cache" ref="Cache"/>
  <property name="Name" value="exampleregion"/>
  <property name="ClientNotification" value="true"/>
  <property name="Interests">
    <list>
      <object type="Spring.Data.GemFire.RegexInterest">
        <property name="Regex" value="Key-.*"/>
      </object>
    </list>
  </property>
</object>
```

This would only register interest in keys of the type 'Key-123' or 'Key-4abc'.

Note

For the M2 release the use of namespaces and type converters for 'ICacheableKey' subclasses will make this configuration significantly less verbose. Also, there will a means to specify the collection class that can be filled with the intial cache values.

Please refer to the API documentation for more information on the various IInterest subclasses.

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 DaoException hierarchy.

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 exceptions to a set of focused data access 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]` attribute and AOP by defining a PersistenceExceptionTranslationPostProcessor object. The same exception translation functionality is enabled when using GemFire as long as at least a CacheFactoryObject is declared. The Cache factory acts as an exception translator which is automatically detected by the Spring infrastructure and used accordingly.

Chapter 3. 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 for running 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 included in the main solution file. You can also run it from the command line once it is built in visual studio.

3.1. Prerequisites

1. You will need to download, install, and obtain a license for

- GemFire Enterprise 6.0
- GemFire Enterprise Native Client 3.0.0.9

The GemFire Enterprise licence file, gemfireLicense.zip should reside in the root of your GemFire Enterprise install directory. The GemFire Enterprise Native Client licence file, gfCpplicense.zip, should reside in the 'bin' sub-directory of the Native Client install directory.

The .NET clients run in a client-server architecture. This is shown below from the perspective of a single 'native (C++ or .NET) client and a single cache server process.

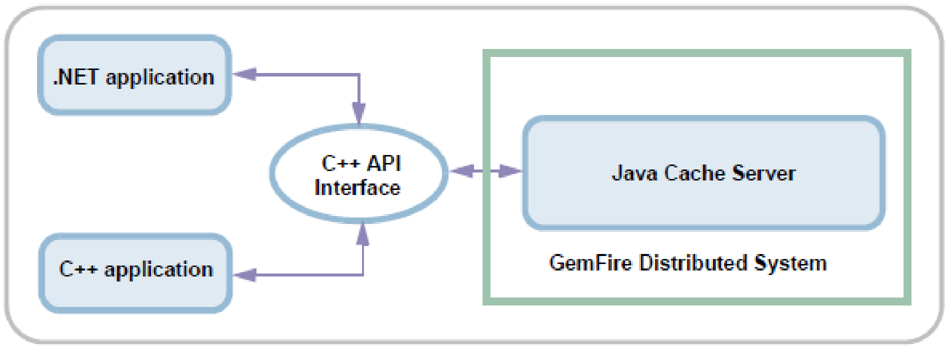

Figure 3.1. Client-Server Configuration for native (C++ or .NET) clients

While not shown in this picture, there can be multiple .NET/C++ client applications and also multiple other processes that act in a peer-to-peer like manner as part of the GemFire Distributed System.

When running in a client-server architecture there needs to be configuration of the client side, the .NET/C++ Application, and the data grid side, the Java Cache Server. In this example the configuration of the client side is done entirely by the FactoryObject's declared in the Spring XML file. The data grid side is configured with a standalone configuration file.

2. The Java Cache Server requires a configuration file in order to run. This file is provided in the Spring GemFire distribution and in named `cache.xml`. It is located in the 'example\Spring.Data.Gemfire.HelloWorld' directory. It needs to be into the GemFire Enterprise 6.0 'bin' directory where the cacheserver.bat file is located.

3. Run the cacheserver.bat file located in the 'bin' of the GemFire Enterprise 6.0 product. By default it will load a configuration file named '`cache.xml`'.

The Spring GemFire project ships with the essential GemFire client side libraries to run the sample application. Note that because the .NET Client API is a wrapper around the C++ API, it needs to be referenced in either the App.config file or installed into the GAC. The HelloWorld example program is configured to load the libraries from the 'lib\GemFire\net\2.0' directory.

Despite providing the client However, it is recommended that you download the GemFire Enterprise Native Client 3.0.0.9 from the download site to have access to reference documentation. You may also find the GemFire .NET API Tour of interest to read.

3.2. Hello World

The Hello World sample demonstrates the basic functionality of the Spring GemFire project and is also useful to understand how GemFire clients work. The application bootstraps GemFire, configures it, allows for the execution of several commands against the data grid, and gracefully shuts 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.

3.2.1. Compiling, starting and stopping the sample

Hello World is designed as a stand-alone application. The main class is in the file `Program.cs` and generates an executable named `HelloWorld.exe`. To start the example follow the steps

To compile the example, load the solution `Spring.Data.GemFire.sln` in the root of the Spring GemFire project directory.

1. Load the Visual Studio 2008 solution Spring.Data.GemFire.sln located in the root of the Spring GemFire project directory. Compile the solution.

2. Ensure that the Java Cache Server is running as described in the previous section.

3. Set the startup project to be the Spring.Data.Gemfire.HelloWorld project **or** cd to the '`example\Spring.Data.Gemfire.HelloWorld\bin\Debug`' directory and run `HelloWorld.exe`.

   Note

   You can pass as a command line argument the name that will appear before you enter commands in the shell. This is useful for distinguishing different members of the distributed system

4. You can now execute commands in the shell, which will be described in the next section. Exit the shell by typing '`exit`'.

3.2.2. Using the sample

Once started, the sample will create a client side cache that is replicated with the server cache contained in the Java Cache Server. For example, the command line '`HelloWorld.exe client-1`' will result in the following greeting

```
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
size - returns the size of the grid
clear - removes all mapping in the grid
```

```
keys - returns the keys contained by the grid
values - returns the values contained by the grid
containsKey <key> - indicates if the given key is contained by the grid
containsValue <value> - indicates if the given value is contained by the grid
map - returns a list of the key-value pairs in the grid

query <query> - executes a query on the grid

help - this info
exit - this node exists
client-1>
```

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

```
client-1>put 1 unu
null
client-1>put 1 one
old value = [unu]
client-1>size
1
client-1>put 2 two
null
client-1>size
2
client-1>
```

Multiple instances can be created at the same time. Once started, the new clients automatically see the existing region and its information. Start a second client with the command line '`HelloWorld.exe client-2`'

```
Hello World!
...

client-2>size
2
client-2>map
[2=two][1=one]
client-2>
```

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, the Java Cache Servier needs to be running at all times.

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 4. Useful Links

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