Example 13-41. Converting a FieldSet to a Product domain object

```
public class ProductFieldSetMapper implements FieldSetMapper<Product> {

  public Product mapFieldSet(FieldSet fieldSet) {
    Product product = new Product();
    product.setId(fieldSet.readString("id"));
    product.setName(fieldSet.readString("name"));
    product.setDescription(fieldSet.readString("description"));
    product.setPrice(fieldSet.readBigDecimal("price"));
    return product;
  }
}
```

The two parts that remain to be configured are the `ItemProcessor` and `ItemWriter`; they are shown in Example 13-42.

Example 13-42. Configuration of a Spring Batch item process and JDBC writer

```
<bean id="processor" class="com.oreilly.springdata.batch.item.ProductProcessor"/>

<bean id="jdbcWriter" class="org.springframework.batch.item.database.JdbcBatchItemWriter">
  <property name="dataSource" ref="dataSource"/>
  <property name="sql"
            value="INSERT INTO PRODUCT (ID, NAME, PRICE) VALUES (:id, :name, :price)"/>
  <property name="itemSqlParameterSourceProvider">
    <bean class="org.sfw.batch.item.database.BeanPropertyItemSqlParameterSourceProvider"/>
  </property>
</bean>
```

`ItemProcessor`s are commonly used to transform, filter, or validate the data. In Example 13-43, we use a simple filter that will pass only through records whose product description ID starts with PR1. Returning a `null` value from an `ItemProcessor` is the contract to filter out the item. Note that if you do not want to perform any processing and directly copy the input file to the database, you can simply remove the processor attribute from the tasklet's chunk XML configuration.

Example 13-43. A simple filtering ItemProcessor

```
public class ProductProcessor implements ItemProcessor<Product, Product> {

  @Override
  public Product process(Product product) throws Exception {
    if (product.getId().startsWith("PR1")) {
      return null;
    } else {
      return product;
    }
  }
}
```

The `JdbcBatchItemWriter` groups together a batch of SQL statements to commit together to the database. The batch size is equal to the commit interval defined previously.

We connect to the database using a standard JDBC `DataSource`, and the SQL statement is specified inline. What is nice about the SQL statement is that we can use named parameters instead of positional `?` placeholders. This is a feature provided by `BeanPropertyItemSqlParameterSourceProvider`, which associates the names of properties of the `Product` object with the `:name` values inside the SQL statement. Example 13-44 uses the H2 database and the schema for the product table.

Example 13-44. Schema definition of the product table

```
create table product (
  id character(9) not null,
  name character varying(50),
  description character varying(255),
  price float,
  update_timestamp timestamp,
  constraint product_pkey primary key (id)
);
```

To start the database, copy the sample data into HDFS, create the Spring Batch schema, and execute the commands shown in Example 13-45. Running these commands will also launch the H2 interactive web console.

Example 13-45. Building the example and starting the database

```
$ cd hadoop/batch-extract
$ mvn clean package appassembler:assemble
$ sh ./target/appassembler/bin/start-database &
```

Next run the export process, as shown in Example 13-46.

Example 13-46. Running the export job

```
$ sh ./target/appassembler/bin/export
INFO - Loaded JDBC driver: org.h2.Driver
INFO - Established shared JDBC Connection: conn0: \
url=jdbc:h2:tcp://localhost/mem:hadoop_export user=SA
INFO - No database type set, using meta data indicating: H2
INFO - No TaskExecutor has been set, defaulting to synchronous executor.
INFO - Job: [FlowJob: [name=exportProducts]] launched with the following parameters: \
       [{hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}]
INFO - Executing step: [readWriteProducts]
INFO - Job: [FlowJob: [name=exportProducts]] completed with the following parameters: \
       [{hdfsSourceDirectory=/data/analysis/results/part-*, date=1345578343793}] and the \
       following status: [COMPLETED]
```

We can view the imported data using the H2 web console. Spring Batch also has an administrative console, where you can browse what jobs are available to be run, as well as look at the status of each job execution. To launch the administrative console, run **`sh ./target/appassembler/bin/launchSpringBatchAdmin`**, open a browser to *http://localhost:8080/springbatchadmin/jobs/executions*, and select the recent job execution link from the table. By clicking on the link for a specific job execution, you can view details about the state of the job.

From HDFS to MongoDB

To write to MongoDB instead of a relational database, we need to change the `ItemWriter` implementation from `JdbcBatchItemWriter` to `MongoItemWriter`. Example 13-47 shows a simple implementation of `MongoItemWriter` that writes the list of items using MongoDB's batch functionality, which inserts the items into the collection by making a single call to the database.

Example 13-47. An ItemWriter implementation that writes to MongoDB

```
public class MongoItemWriter implements ItemWriter<Object> {

  private MongoOperations mongoOperations;
  private String collectionName = "/data";

  // constructor and setters omitted.

  @Override
  public void write(List<? extends Object> items) throws Exception {
    mongoOperations.insert(items, collectionName);
  }
}
```

Spring's `MongoTemplate` (which implements the interface `MongoOperations`) provides support for converting Java classes to MongoDB's internal data structure format, `DbObject`. We can specify the connectivity to MongoDB using the Mongo XML namespace. Example 13-48 shows the configuration of `MongoItemWriter` and its underlying dependencies to connect to MongoDB.

Example 13-48. Configuration of a Spring Batch job to read from HDFS and write to MongoDB

```
<job id="exportProducts">
  <step id="readWriteProducts">
    <tasklet>
      <chunk reader="reader" writer="mongoWriter" commit-interval="100" skip-limit="5">
        <skippable-exception-classes>
          <include class="org.springframework.batch.item.file.FlatFileParseException"/>
        </skippable-exception-classes>
      </chunk>
    </tasklet>
  </step>
</job>

<!-- reader configuration is the same as before -->

<bean id="mongoWriter" class="com.oreilly.springdata.batch.item.mongodb.MongoItemWriter">
  <constructor-arg ref="mongoTemplate"/>
  <property name="collectionName" value="products"/>
</bean>

<bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
    <constructor-arg ref="mongo"/>
    <constructor-arg name="databaseName" value="test"/>
```

```
</bean>

<mongo:mongo host="localhost" port="27017"/>
```

Running the application again will produce the contents of the products collection in the test database, shown in Example 13-49.

Example 13-49. Viewing the exported products collection in the MongoDB shell

```
$ mongo
> use test
> db.products.find()
{"_id" : "PR 210", "PRICE" : "124.60", "NAME" : "BlackBerry 8100 Pearl",
    "DESCRIPTION" : ""}
{"_id" : "PR 211", "PRICE" : "139.45", "NAME" : "Sony Ericsson W810i", "DESCRIPTION" : ""}
{"_id" : "PR 212", "PRICE" : "97.80", "NAME" : "Samsung MM-A900M Ace", "DESCRIPTION" : ""}

(output truncated)
```

This example shows how various Spring Data projects build upon each other to create important new features with a minimal amount of code. Following the same pattern to implement a `MongoItemWriter`, you can also easily create `ItemWriter`s for Redis or GemFire that would be as simple as the code shown in this section.

Collecting and Loading Data into Splunk

Splunk collects, indexes, searches, and monitors large amounts of machine-generated data. Splunk can process streams of real-time data as well as historical data. The first version of Splunk was released in 2005, with a focus on analyzing the data generated inside of datacenters to help solve operational infrastructure problems. As such, one of its core capabilities is moving data generated on individual machines into a central repository, as well as having out-of-the box knowledge of popular log file formats and infrastructure software like syslog. Splunk's base architecture consists of a splunkd daemon that processes and indexes streaming data, and a web application that allows users to search and create reports. Splunk can scale out by adding separate indexer, search, and forwarder instances as your data requirements grow. For more details on how to install, run, and develop with Splunk, refer to the product website, as well as the book *Exploring Splunk*.

While the process of collecting log files and syslog data are supported out-of-the-box by Splunk, there is still a need to collect, transform, and load data into Splunk that comes from a variety of other sources, in order to reduce the need to use regular expressions when analyzing data. There is also a need to transform and extract data out of Splunk into other databases and filesystems. To address these needs, Spring Integration inbound and outbound channel adapters were created, and Spring Batch support is on the road map. At the time of this writing, the Spring Integration channel adapters for Splunk are located in the GitHub repository for Spring Integration extensions. The adapters support all the ways you can get data in and out of Splunk. The

inbound adapter supports block and nonblocking searches, saved and real time searches, and the export mode. The outbound adapters support putting data into Splunk through its RESTful API, streams, or TCP. All of the functionality of Splunk is exposed via a comprehensive REST API, and several language SDKs are available to make developing with the REST API as simple as possible. The Spring Integration adapters make use of the Splunk Java SDK , also available on Github.

As an introduction to using Splunk with Spring Integration, we will create an application that stores the results from a Twitter search into Splunk. The configuration for this application is shown in Example 13-50.

Example 13-50. Configuring an application to store Twitter search results in Splunk

```
<context:property-placeholder location="twitter.properties,splunk.properties" />

<bean id="twitterTemplate"
  class="org.springframework.social.twitter.api.impl.TwitterTemplate">
  <constructor-arg value="${twitter.oauth.consumerKey}" />
  <constructor-arg value="${twitter.oauth.consumerSecret}" />
  <constructor-arg value="${twitter.oauth.accessToken}" />
  <constructor-arg value="${twitter.oauth.accessTokenSecret}" />
</bean>

<int-splunk:server id="splunkServer"
                   host="${splunk.host}" port="${splunk.port}"
                   userName="${splunk.userName}" password="${splunk.password}"
                   owner="${splunk.owner}"/>

<int:channel id="input"/>

<int:channel id="output"/>

<int-twitter:search-inbound-channel-adapter id="searchAdapter" channel="input"
             query="#voteobama OR #voteromney OR #votebieber">
  <int:poller fixed-rate="5000" max-messages-per-poll="50" />
</int-twitter:search-inbound-channel-adapter>

<int:chain input-channel="input" output-channel="output">
  <int:filter ref="tweetFilter"/>
  <int:transformer ref="splunkTransformer"/>
</int:chain>

<int-splunk:outbound-channel-adapter id="splunkOutboundChannelAdapter"
            channel="output" auto-startup="true"
            splunk-server-ref="splunkServer" pool-server-connection="true"
            sourceType="twitter-integration" source="twitter" ingest="SUBMIT"/>

<!-- tweetFilter and splunkTransformer bean definitions omitted -->
```

The Spring Social project provides the foundation for connecting to Twitter with OAuth, and also provides support for working with Facebook, LinkedIn, TripIt, Foursquare, and dozens of other social Software-as-a-Service providers. Spring Social's `TwitterTemplate` class is used by the inbound channel adapter to interact with Twitter.

This requires you to create a new application on the Twitter Developer website, in order to access the full range of functionality offered by Twitter. To connect to the Splunk server, the XML namepace `<int-splunk:server/>` is used, providing host/port information as well as user credentials (which are externally parameterized using Spring's property placeholder).

The Twitter inbound channel adapter provides support for receiving tweets as Timeline Updates, Direct Messages, Mention Messages, or Search Results. Note that there will soon be support for consuming data from the Twitter garden hose, which provides a randomly selected stream of data capped at a small percent of the full stream. In Example 13-50, the query looks for hashtags related to the United States 2012 presidental election and one hashtag to vote for Justin Bieber to win various award shows. The processing chain applies a filter and then converts the Tweet payload object into a data structure with a format optimized to help Splunk index and search the tweets. In the sample application, the filter is set to be a pass-through. The outbound channel adapter writes to the Splunk source with the REST API, specified by the attribute `inject="SUBMIT"`. The data is written to the Splunk source named *twitter* and uses the default index. You can also set the `index` attribute to specify that data should be written to the non-default index.

To run the example and see who is the most popular candidate (or pseudo-candidate), follow the directions in the directory *splunk/tweets*. Then via the Splunk web application you can create a search, such as `source="twitter" | regex tags="^voteobama$|^votebieber$|^voteromney$" | top tags`, to get a graph that shows the relative popularity of those individual hashtags.

PART VI

Data Grids

CHAPTER 14

GemFire: A Distributed Data Grid

vFabric™GemFire® (GemFire) is a commercially licensed data management platform that provides access to data throughout widely distributed architectures. It is available as a standalone product and as a component of the VMware vFabric Suite. This chapter provides an overview of Spring Data GemFire. We'll begin by introducing GemFire and some basic concepts that are prerequisite to developing with GemFire. Feel free to skip to the section "Configuring GemFire with the Spring XML Namespace" on page 258 if you are already familiar with GemFire.

GemFire in a Nutshell

GemFire provides an in-memory data grid that offers extremely high throughput, low latency data access, and scalability. Beyond a distributed cache, GemFire provides advanced features including:

- Event notification
- OQL (Object Query Language) query syntax
- Continuous queries
- Transaction support
- Remote function execution
- WAN communications
- Efficient and portable object serialization (PDX)
- Tools to aid system administrators in managing and configuring the GemFire distributed system

GemFire may be configured to support a number of distributed system topologies and is completely integrated with the Spring Framework. Figure 14-1 shows a typical client server configuration for a production LAN. The locator acts as a broker for the distributed system to support discovery of new member nodes. Client applications use the locator to acquire connections to cache servers. Additionally, server nodes use the

locator to discover each other. Once a server comes online, it communicates directly with its peers. Likewise, once a client is initialized, it communicates directly with cache servers. Since a locator is a single point of failure, two instances are required for redundancy.

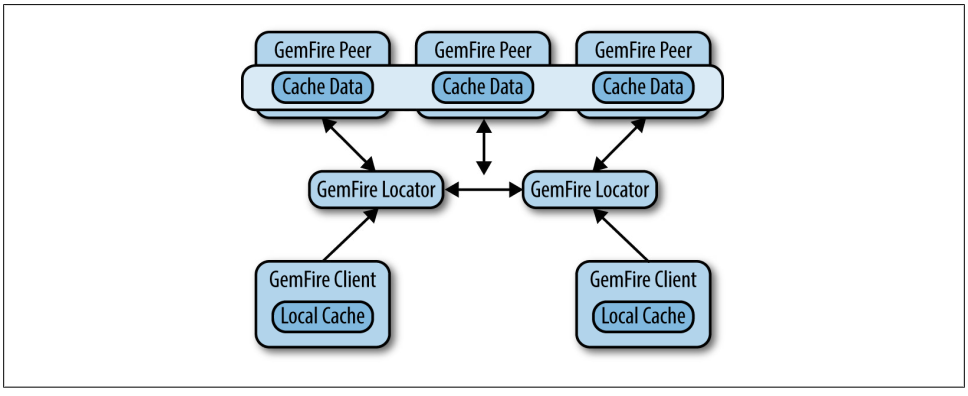

Figure 14-1. GemFire client server topology

Simple standalone configurations for GemFire are also possible. Note that the book's code samples are configured very simply as a single process with an embedded cache, suitable for development and integration testing.

In a client server scenario, the application process uses a *connection pool* (Figure 14-2) to manage connections between the client cache and the servers. The connection pool manages network connections, allocates threads, and provides a number of tuning options to balance resource usage and performance. The pool is typically configured with the address of the locator(s) [not shown in Figure 14-2]. Once the locator provides a server connection, the client communicates directly with the server. If the primary server becomes unavailable, the pool will acquire a connection to an alternate server if one is available.

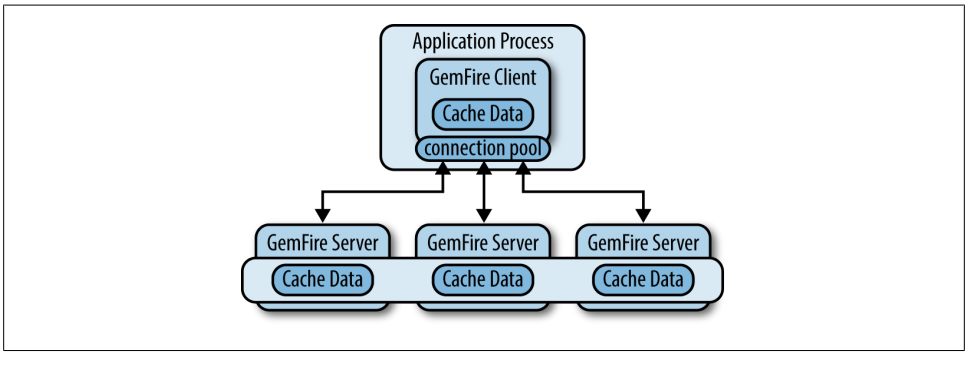

Figure 14-2. The connection pool

Caches and Regions

Conceptually, a *cache* is a singleton object that provides access to a GemFire member and offers a number of configuration options for memory tuning, network communications, and other features. The cache also acts as a container for *regions*, which provide data management and access.

A region is required to store and retrieve data from the cache. `Region` is an interface that extends `java.uti.Map` to perform basic data access using familiar key/value semantics. The `Region` interface is wired into classes that require it, so the actual region type is decoupled from the programming model (with some caveats, the discovery of which will be left as an exercise for the reader). Typically, each region is associated with one domain object, similar to a table in a relational database. Looking at the sample code, you will see three regions defined: `Customer`, `Product`, and `Order`. Note that GemFire does not manage associations or enforce relational integrity among regions.

GemFire includes the following types of regions:

Replicated
: Data is replicated across all cache members that define the region. This provides very high read performance, but writes take longer due to the need to perform the replication.

Partitioned
: Data is partitioned into buckets among cache members that define the region. This provides high read and write performance and is suitable for very large datasets that are too big for a single node.

Local
: Data exists only on the local node.

Client
: Technically, a client region is a local region that acts as a proxy to a replicated or partitioned region hosted on cache servers. It may hold data created or fetched locally; alternatively, it can be empty. Local updates are synchronized to the cache server. Also, a client region may subscribe to events in order to stay synchronized with changes originating from remote processes that access the same region.

Hopefully, this brief overview gives you a sense of GemFire's flexibility and maturity. A complete discussion of GemFire options and features is beyond the scope of this book. Interested readers will find more details on the product website.

How to Get GemFire

The vFabric GemFire website provides detailed product information, reference guides, and a link to a free developer download, limited to three node connections. For a more comprehensive evaluation, a 60-day trial version is also available.

The product download is not required to run the code samples included with this book. The GemFire jar file that includes the free developer license is available in public repositories and will be automatically downloaded by build tools such as Maven and Gradle when you declare a dependency on Spring Data GemFire. A full product install is necessary to use locators, the management tools, and so on.

Configuring GemFire with the Spring XML Namespace

Spring Data GemFire includes a dedicated XML namespace to allow full configuration of the data grid. In fact, the Spring namespace is considered the preferred way to configure GemFire, replacing GemFire's native *cache.xml* file. GemFire will continue to support *cache.xml* for legacy reasons, but you can now do everything in Spring XML and take advantage of the many wonderful things Spring has to offer, such as modular XML configuration, property placeholders, SpEL, and environment profiles. Behind the namespace, Spring Data GemFire makes extensive use of Spring's `FactoryBean` pattern to simplify the creation and initialization of GemFire components.

GemFire provides several callback interfaces, such as `CacheListener`, `CacheWriter`, and `CacheLoader` to allow developers to add custom event handlers. Using the Spring IoC container, these may configured as normal Spring beans and injected into GemFire components. This is a significant improvement over *cache.xml*, which provides relatively limited configuration options and requires callbacks to implement GemFire's `Declarable` interface.

In addition, IDEs such as the Spring Tool Suite (STS) provide excellent support for XML namespaces, such as code completion, pop-up annotations, and real-time validation, making them easy to use.

The following sections are intended to get you started using the Spring XML namespace for GemFire. For a more comprehensive discussion, please refer to the Spring Data GemFire reference guide at the project website.

Cache Configuration

To configure a GemFire cache, create a Spring bean definition file and add the Spring GemFire namespace. In STS (Figure 14-3), select the project and open the context menu (right-click) and select New→Spring Bean Configuration File. Give it a name and click Next.

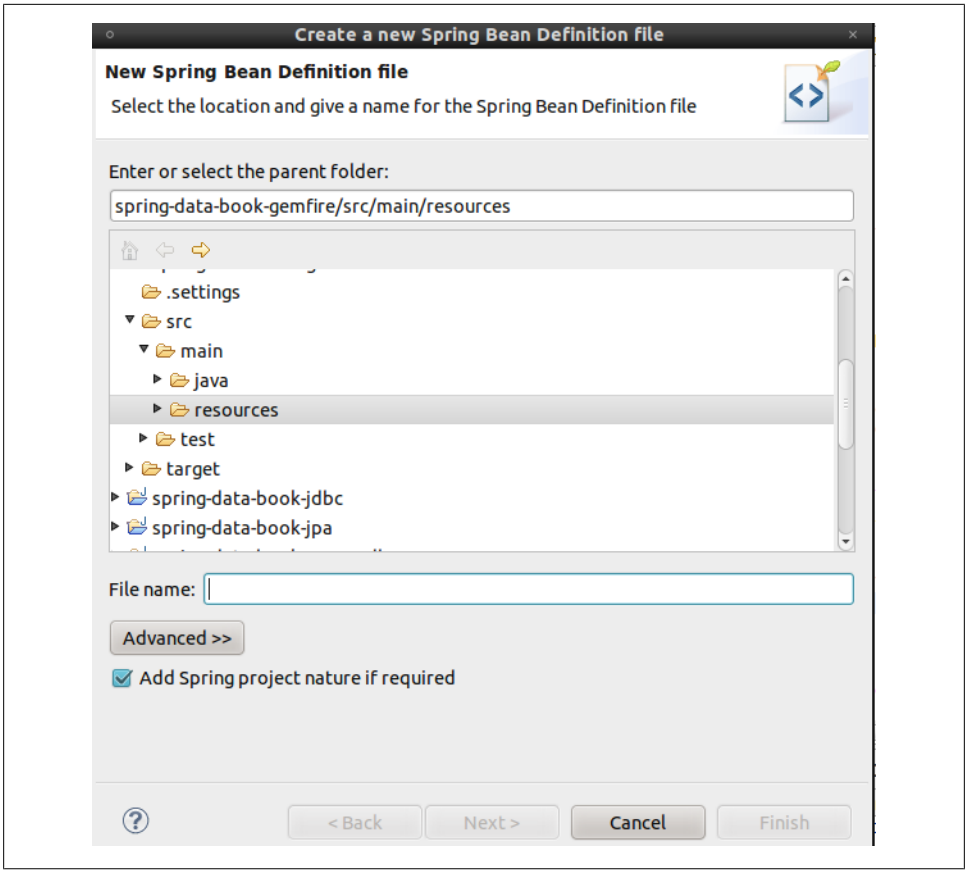

Figure 14-3. Create a Spring bean definition file in STS

In the XSD namespaces view, select the `gfe` namespace (Figure 14-4).

Notice that, in addition to the `gfe` namespace, there is a `gfe-data` namespace for Spring Data POJO mapping and repository support. The `gfe` namespace is used for core GemFire configuration.

Click Finish to open the bean definition file in an XML editor with the correct namespace declarations. (See Example 14-1.)

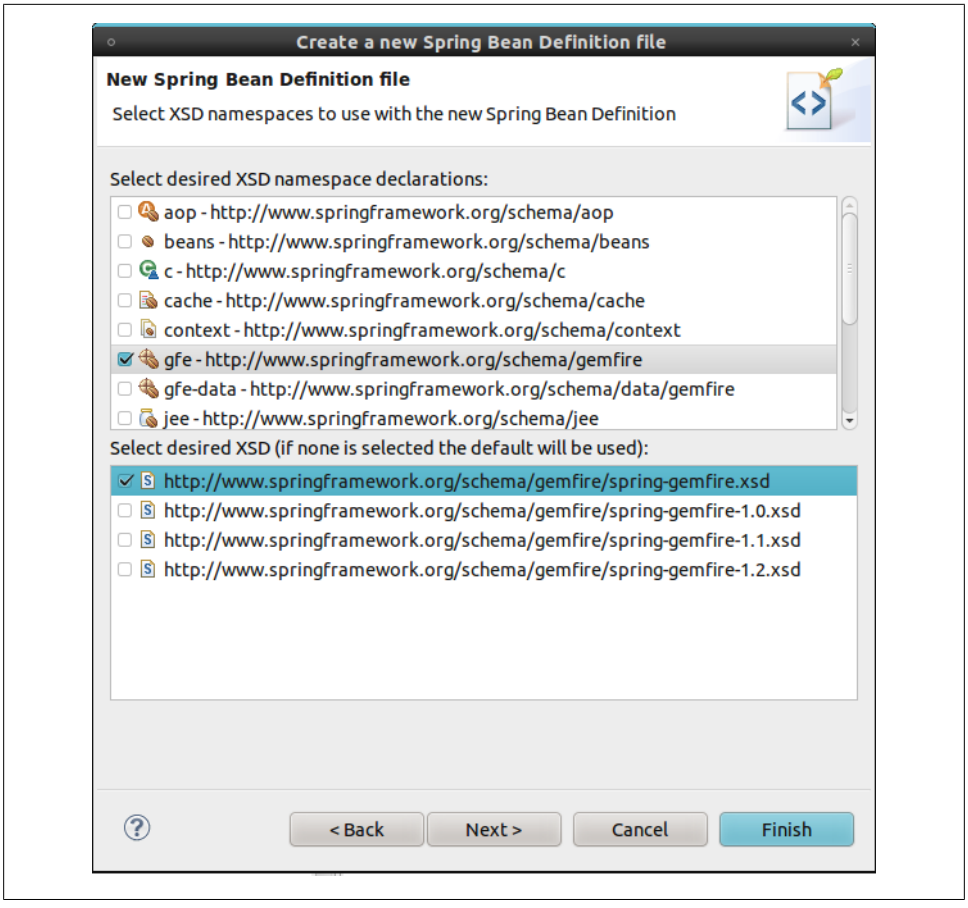

Figure 14-4. Selecting Spring XML namespaces

Example 14-1. Declaring a GemFire cache in Spring configuration

```
<?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:gfe="http://www.springframework.org/schema/gemfire"
       xsi:schemaLocation="http://www.springframework.org/schema/gemfire
                           http://www.springframework.org/schema/gemfire/spring-gemfire.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

   <gfe:cache/>

</beans>
```

Now use the `gfe` namespace to add a `cache` element. That's it! This simple cache declaration will create an embedded cache, register it in the Spring `ApplicationContext` as `gemfireCache`, and initialize it when the context is created.

Prior releases of Spring Data GemFire created default bean names using hyphens (e.g., `gemfire-cache`). As of the 1.2.0 release, these are replaced with camelCase names to enable autowiring via annotations (`@Autowired`). The old-style names are registered as aliases to provide backward compatibility.

You can easily change the bean name by setting the `id` attribute on the `cache` element. However, all other namespace elements assume the default name unless explicitly overridden using the `cache-ref` attribute. So you can save yourself some work by following the convention.

The `cache` element provides some additional attributes, which STS will happily suggest if you press Ctrl-Space. The most significant is the `properties-ref` attribute. In addition to the API, GemFire exposes a number of global configuration options via external properties. By default, GemFire looks for a file called *gemfire.properties* in all the usual places: the user's home directory, the current directory, and the classpath. While this is convenient, it may result in unintended consequences if you happen to have these files laying around. Spring alleviates this problem by offering several better alternatives via its standard property loading mechanisms. For example, you can simply construct a `java.util.Properties` object inline or load it from a properties file located on the classpath or filesystem. Example 14-2 uses properties to configure GemFire logging.

Example 14-2. Referencing properties to configure GemFire

Define properties inline:

```
<util:properties id="props">
   <prop key="log-level">info</prop>
   <prop key="log-file">gemfire.log</prop>
</util:properties>

<gfe:cache properties-ref="props" />
```

Or reference a resource location:

```
<util:properties id="props" location="gemfire-cache.properties" />

<gfe:cache properties-ref="props" />
```

It is generally preferable to maintain properties of interest to system administrators in an agreed-upon location in the filesystem rather than defining them in Spring XML or packaging them in *.jar* files.

Note the use of Spring's `util` namespace to create a `Properties` object. This is related to, but not the same as, Spring's property placeholder mechanism, which uses token-based substitution to allow properties on any bean to be defined externally from a variety of sources. Additionally, the `cache` element includes a `cache-xml-location` attribute to enable the cache to be configured with GemFire's native configuration schema. As previously noted, this is mostly there for legacy reasons.

The cache element also provides some `pdx-*` attributes required to enable and configure GemFire's proprietary serialization feature (PDX). We will address PDX in "Repository Usage" on page 271.

For advanced cache configuration, the cache element provides additional attributes for tuning memory and network communications (shown in Figure 14-5) and child elements to register callbacks such as `TransactionListers`, `and TransactionWriters` (Figure 14-6).

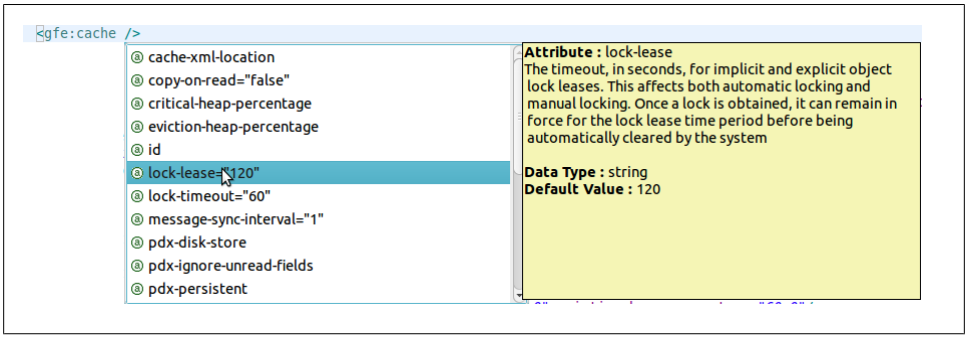

Figure 14-5. Displaying a list of cache attributes in STS

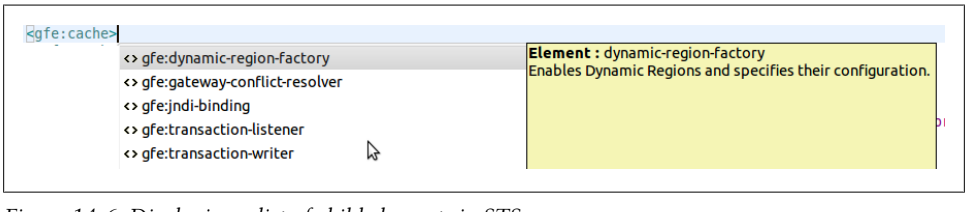

Figure 14-6. Displaying a list of child elements in STS

The `use-bean-factory-locator` attribute (not shown) deserves a mention. The factory bean responsible for creating the cache uses an internal Spring type called a `BeanFactoryLocator` to enable user classes declared in GemFire's native *cache.xml* file to be registered as Spring beans. The `BeanFactoryLocator` implementation also permits only one bean definition for a cache with a given `id`. In certain situations, such as running JUnit integration tests from within Eclipse, you'll need to disable the `BeanFactoryLocator` by setting this value to `false` to prevent an exception. This exception may also arise during JUnit tests running from a build script. In this case, the test runner should be configured to fork a new JVM for each test (in Maven, set `<forkmode>always</forkmode>`). Generally, there is no harm in setting this value to `false`.

Region Configuration

As mentioned in the chapter opener, GemFire provides a few types of regions. The XML namespace defines `replicated-region`, `partitioned-region`, `local-region`, and `client-region` elements to create regions. Again, this does not cover all available features but highlights some of the more common ones. A simple region declaration, as shown in Example 14-3, is all you need to get started.

Example 14-3. Basic region declaration

```
<gfe:cache/>
<gfe:replicated-region id="Customer" />
```

The region has a dependency on the cache. Internally, the cache creates the region. By convention, the namespace does the wiring implicitly. The default cache declaration creates a Spring bean named `gemfireCache`. The default region declaration uses the same convention. In other words, Example 14-3 is equivalent to:

```
<gfe:cache id="gemfireCache" />
<gfe:replicated-region id="Customer" cache-ref="gemfireCache" />
```

If you prefer, you can supply any valid bean name, but be sure to set `cache-ref` to the corresponding bean name as required.

Typically, GemFire is deployed as a distributed data grid, hosting replicated or partitioned regions on cache servers. Client applications use client regions to access data. For development and integration testing, it is a best practice to eliminate any dependencies on an external runtime environment. You can do this by simply declaring a replicated or local region with an embedded cache, as is done in the sample code. Spring environment profiles are extremely useful in configuring GemFire for different environments.

In Example 14-4, the `dev` profile is intended for integration testing, and the `prod` profile is used for the deployed cache configuration. The cache and region configuration is transparent to the application code. Also note the use of property placeholders to

specify the locator hosts and ports from an external properties file. Cache client configuration is discussed further in "Cache Client Configuration" on page 265.

Example 14-4. Sample XML configuration for development and production

```
<beans profile="dev">
  <gfe:cache/>
  <gfe:replicated-region id="Customer" />
</beans>

<beans profile="prod">
  <context:properties-placeholder location="client-app.properties" />
  <gfe:client-cache pool-name="pool" />

  <gfe:client-region id="Customer" />

  <gfe:pool id="pool">
    <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
    <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
 </gfe:pool>
</beans>
```

Spring provides a few ways to activate the appropriate environment profile(s). You can set the property `spring.profiles.active` in a system property, a servlet context parameter, or via the `@ActiveProfiles` annotation.

As shown in Figure 14-7, there are a number of common region configuration options as well as specific options for each type of region. For example, you can configure all regions to back up data to a local disk store synchronously or asynchronously.

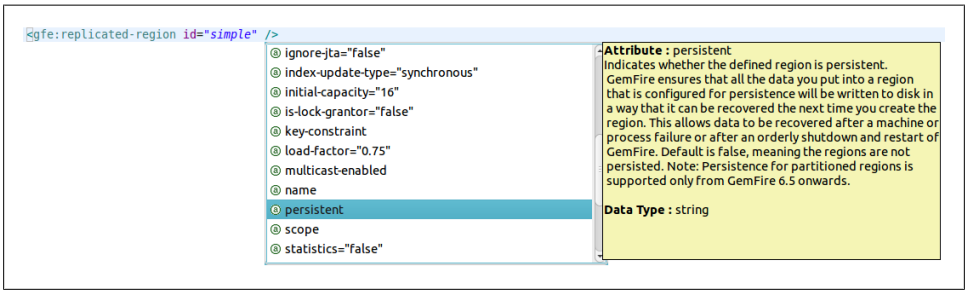

Figure 14-7. Displaying replicated region attributes in STS

Additionally, you may configure regions to synchronize selected entries over a WAN gateway to distribute data over a wide geographic area. You may also register `CacheListeners`, `CacheLoaders`, and `CacheWriters` to handle region events. Each of these interfaces is used to implement a callback that gets invoked accordingly. A `CacheListener` is a generic event handler invoked whenever an entry is created, updated,

destroyed, etc. For example, you can write a simple `CacheListener` to log cache events, which is particularly useful in a distributed environment (see Example 14-5). A `CacheLoader` is invoked whenever there is a cache miss (i.e., the requested entry does not exist), allowing you to "read through" to a database or other system resource. A `CacheWriter` is invoked whenever an entry is updated or created to provide "write through" or "write behind" capabilities.

Example 14-5. LoggingCacheListener implementation

```
public class LoggingCacheListener extends CacheListenerAdapter {

  private static Log log = LogFactory.getLog(LoggingCacheListener.class);

  @Override
  public void afterCreate(EntryEvent event) {
     String regionName = event.getRegion().getName();
     Object key = event.getKey();
     Object newValue = event.getNewValue();
     log.info("In region [" + regionName + "] created key ["
       + key + "] value [" + newValue + "]");
   }

   @Override
   public void afterDestroy(EntryEvent event) {
    …
   }

   @Override
   public void afterUpdate(EntryEvent event) {
    …
   }
}
```

Other options include *expiration*, the maximum time a region or an entry is held in the cache, and *eviction*, policies that determine which items are removed from the cache when the defined memory limit or the maximum number of entries is reached. Evicted entries may optionally be stored in a disk overflow.

You can configure partitioned regions to limit the amount of local memory allocated to each partition node, define the number of buckets used, and more. You may even implement your own `PartitionResolver` to control how data is colocated in partition nodes.

Cache Client Configuration

In a client server configuration, application processes are *cache clients*—that is, they produce and consume data but do not distribute it directly to other processes. Neither does a cache client implicitly see updates performed by remote processes. As you might expect by now, this is entirely configurable. Example 14-6 shows a basic client-side setup using a `client-cache`, `client-region`, and a `pool`. The `client-cache` is a

lightweight implementation optimized for client-side services, such as managing one or more client regions. The `pool` represents a connection pool acting as a bridge to the distributed system and is configured with any number of locators.

Typically, two locators are sufficient: the first locator is primary, and the remaining ones are strictly for failover. Every distributed system member should use the same locator configuration. A locator is a separate process, running in a dedicated JVM, but is not strictly required. For development and testing, the pool also provides a `server` child element to access cache servers directly. This is useful for setting up a simple client/server environment (e.g., on your local machine) but not recommended for production systems. As mentioned in the chapter opener, using a locator requires a full GemFire installation, whereas you can connect to a server directly just using the APIs provided in the publicly available *gemfire.jar* for development, which supports up to three cache members.

Example 14-6. Configuring a cache pool

```
<gfe:client-cache pool-name="pool" />

<gfe:client-region id="Customer" />

<gfe:pool id="pool">
  <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
  <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

You can configure the `pool` to control thread allocation for connections and network communications. Of note is the `subscription-enabled` attribute, which you must set to `true` to enable synchronizing region entry events originating from remote processes (Example 14-7).

Example 14-7. Enabling subscriptions on a cache pool

```
<gfe:client-region id="Customer">
   <gfe:key-interest durable="false" receive-values="true" />
</client-region>

<gfe:pool id="pool" subcription-enabled="true">
  <gfe:locator host="${locator.host.1}" port="${locator.port.1}"/>
  <gfe:locator host="${locator.host.2}" port="${locator.port.2}"/>
</gfe:pool>
```

With subscriptions enabled, the `client-region` may register interest in all keys or specific keys. The subscription may be durable, meaning that the `client-region` is updated with any events that may have occurred while the client was offline. Also, it is possible to improve performance in some cases by suppressing transmission of values unless

explicitly retrieved. In this case, new keys are visible, but the value must be retrieved explicitly with a `region.get(key)` call, for example.

Cache Server Configuration

Spring also allows you to create and initialize a cache server process simply by declaring the cache and region(s) along with an additional `cache-server` element to address server-side configuration. To start a cache server, simply configure it using the namespace and start the application context, as shown in Example 14-8.

Example 14-8. Bootstrapping a Spring application context

```
public static void main(String args[]) {
   new ClassPathXmlApplicationContext("cache-config.xml");
}
```

Figure 14-8 shows a Spring-configured cache server hosting two partitioned regions and one replicated region. The `cache-server` exposes many parameters to tune network communications, system resources, and the like.

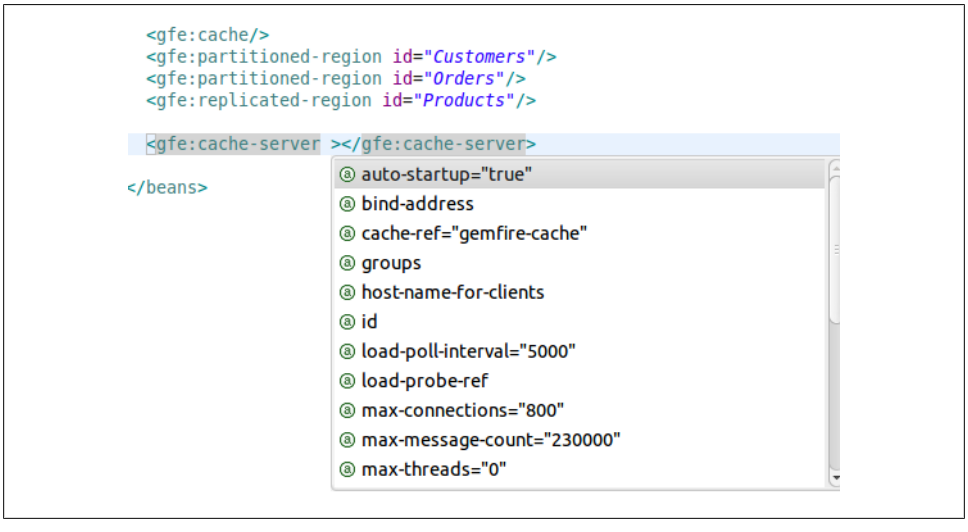

Figure 14-8. Configuring a cache server

WAN Configuration

WAN configuration is required for geographically distributed systems. For example, a global organization may need to share data across the London, Tokyo, and New York offices. Each location manages its transactions locally, but remote locations need to be synchronized. Since WAN communications can be very costly in terms of performance and reliability, GemFire queues events, processed by a WAN gateway to achieve

eventual consistency. It is possible to control which events get synchronized to each remote location. It is also possible to tune the internal queue sizes, synchronization scheduling, persistent backup, and more. While a detailed discussion of GemFire's WAN gateway architecture is beyond the scope of this book, it is important to note that WAN synchronization must be enabled at the region level. See Example 14-9 for a sample configuration.

Example 14-9. GemFire WAN configuration

```
<gfe:replicated-region id="region-with-gateway" enable-gateway="true" hub-id="gateway-hub" />

<gfe:gateway-hub id="gateway-hub" manual-start="true">
  <gfe:gateway gateway-id="gateway">
    <gfe:gateway-listener>
       <bean class="..."/>
    </gfe:gateway-listener>
    <gfe:gateway-queue maximum-queue-memory="5" batch-size="3" batch-time-interval="10" />
  </gfe:gateway>

  <gfe:gateway gateway-id="gateway2">
    <gfe:gateway-endpoint port="1234" host="host1" endpoint-id="endpoint1" />
    <gfe:gateway-endpoint port="2345" host="host2" endpoint-id="endpoint2" />
  </gfe:gateway>
</gfe:gateway-hub>
```

This example shows a region enabled for WAN communications using the APIs available in GemFire 6 versions. The `enable-gateway` attribute must be set to `true` (or is implied by the presence of the `hub-id` attribute), and the `hub-id` must reference a `gateway-hub` element. Here we see the `gateway-hub` configured with two gateways. The first has an optional `GatewayListener` to handle gateway events and configures the gateway queue. The second defines two remote gateway endpoints.

The WAN architecture will be revamped in the upcoming GemFire 7.0 release. This will include new features and APIs, and will generally change the way gateways are configured. Spring Data GemFire is planning a concurrent release that will support all new features introduced in GemFire 7.0. The current WAN architecture will be deprecated.

Disk Store Configuration

GemFire allows you to configure disk stores for persistent backup of regions, disk overflow for evicted cache entries, WAN gateways, and more. Because a disk store may serve multiple purposes, it is defined as a top-level element in the namespace and may be referenced by components that use it. Disk writes may be synchronous or asynchronous.

For asynchronous writes, entries are held in a queue, which is also configurable. Other options control scheduling (e.g., the maximum time that can elapse before a disk write

is performed or the maximum file size in megabytes). In Example 14-10, an overflow disk store is configured to store evicted entries. For asynchronous writes, it will store up to 50 entries in a queue, which will be flushed every 10 seconds or if the queue is at capacity. The region is configured for eviction to occur if the total memory size exceeds 2 GB. A custom `ObjectSizer` is used to estimate memory allocated per entry.

Example 14-10. Disk store configuration

```
<gfe:partitioned-region id="partition-data" persistent="true" disk-store-ref="ds2">
  <gfe:eviction type="MEMORY_SIZE" threshold="2048" action="LOCAL_DESTROY">
  <gfe:object-sizer>
    <bean class="org.springframework.data.gemfire.SimpleObjectSizer" />
  </gfe:object-sizer>
  </gfe:eviction>
</gfe:partitioned-region>

<gfe:disk-store id="ds2" queue-size="50" auto-compact="true"
     max-oplog-size="10" time-interval="10000">
  <gfe:disk-dir location="/gemfire/diskstore" />
</gfe:disk-store>
```

Data Access with GemfireTemplate

Spring Data GemFire provides a template class for data access, similar to the `JdbcTemplate` or `JmsTemplate`. The `GemfireTemplate` wraps a single region and provides simple data access and query methods as well as a callback interface to access region operations. One of the key reasons to use the `GemfireTemplate` is that it performs exception translation from GemFire checked exceptions to Spring's `PersistenceException` runtime exception hierarchy. This simplifies exception handling required by the native `Region` API and allows the template to work more seamlessly with the Spring declarative transactions using the `GemfireTransactionManager` which, like all Spring transaction managers, performs a rollback for runtime exceptions (but not checked exceptions) by default. Exception translation is also possible for `@Repository` components, and transactions will work with `@Transactional` methods that use the `Region` interface directly, but it will require a little more care.

Example 14-11 is a simple demonstration of a data access object wired with the `GemfireTemplate`. Notice the `template.query()` invocation backing `findByLastName(...)`. Queries in GemFire use the Object Query Language (OQL). This method requires only a boolean predicate defining the query criteria. The body of the query, `SELECT * from [region name] WHERE...`, is assumed. The template also implements `find(...)` and `findUnique(...)` methods, which accept parameterized query strings and associated parameters and hide GemFire's underlying `QueryService` API.

Example 14-11. Repository implementation using GemfireTemplate

```
@Repository
class GemfireCustomerRepository implements CustomerRepository {
```

```
  private final GemfireTemplate template;

  @Autowired
  public GemfireCustomerRepository(GemfireTemplate template) {
    Assert.notNull(template);
    this.template = template;
  }

  /**
   * Returns all objects in the region. Not advisable for very large datasets.
   */
  public List<Customer> findAll() {
    return new ArrayList<Customer>((Collection<? extends Customer>) ↪
template.getRegion().values());
  }

  public Customer save(Customer customer) {
    template.put(customer.getId(), customer);
    return customer;
  }

  public List<Customer> findByLastname(String lastname) {

    String queryString = "lastname = '" + lastname + "'";
    SelectResults<Customer> results = template.query(queryString);
    return results.asList();
  }

  public Customer findByEmailAddress(EmailAddress emailAddress) {

    String queryString = "emailAddress = ?1";
    return template.findUnique(queryString, emailAddress);
  }

  public void delete(Customer customer) {
    template.remove(customer.getId());
  }
}
```

We can configure the `GemfireTemplate` as a normal Spring bean, as shown in Example 14-12.

Example 14-12. GemfireTemplate configuration

```
<bean id="template" class="org.springframework.data.gemfire.GemfireTemplate">
  <property name="region" ref="Customer" />
</bean>
```

Repository Usage

The 1.2.0 release of Spring Data GemFire introduces basic support for Spring Data repositories backed by GemFire. All the core repository features described in Chapter 2 are supported, with the exception of paging and sorting. The sample code demonstrates these features.

POJO Mapping

Since GemFire regions require a unique key for each object, the top-level domain objects—`Customer`, `Order`, and `Product`—all inherit from `AbstractPersistentEntity`, which defines an `id` property (Example 14-13).

Example 14-13. AbstractPersistentEntity domain class

```
import org.springframework.data.annotation.Id;

public class AbstractPersistentEntity {

  @Id
  private final Long id;
}
```

Each domain object is annotated with `@Region`. By convention, the region name is the same as the simple class name; however, we can override this by setting the annotation value to the desired region name. This must correspond to the region name—that is, the value of `id` attribute or the `name` attribute, if provided, of the region element. Common attributes, such as `@PersistenceConstructor` (shown in Example 14-14) and `@Transient`, work as expected.

Example 14-14. Product domain class

```
@Region
public class Product extends AbstractPersistentEntity {

  private String name, description;
  private BigDecimal price;
  private Map<String, String> attributes = new HashMap<String, String>();

  @PersistenceConstructor
  public Product(Long id, String name, BigDecimal price, String description) {

    super(id);
    Assert.hasText(name, "Name must not be null or empty!");
    Assert.isTrue(BigDecimal.ZERO.compareTo(price) < 0, "Price must be greater than zero!");

    this.name = name;
    this.price = price;
    this.description = description;
}
```

Creating a Repository

GemFire repositories support basic CRUD and query operations, which we define using Spring Data's common method name query mapping mechanism. In addition, we can configure a repository method to execute any OQL query using `@Query`, as shown in Example 14-15.

Example 14-15. ProductRepository interface

```
public interface ProductRepository extends CrudRepository<Product, Long> {

  List<Product> findByDescriptionContaining(String description);

  /**
   * Returns all {@link Product}s having the given attribute value.
   * @param attribute
   * @param value
   * @return
   */
  @Query("SELECT * FROM /Product where attributes[$1] = $2")
  List<Product> findByAttributes(String key, String value);

  List<Product> findByName(String name);
}
```

You can enable repository discovery using a dedicated `gfe-data` namespace, which is separate from the core `gfe` namespace. Alternatively, if you're using Java configuration, simply annotate your configuration class with `@EnableGemfireRepositories`, as shown in Example 14-16.

Example 14-16. Enabling GemFire repositories using XML

```
<gfe-data:repositories base-package="com.oreilly.springdata.gemfire" />
```

PDX Serialization

PDX is GemFire's proprietary serialization library. It is highly efficient, configurable, interoperable with GemFire client applications written in C# or C++, and supports object versioning. In general, objects must be serialized for operations requiring network transport and disk persistence. Cache entries, if already serialized, are stored in serialized form. This is generally true with a distributed topology. A standalone cache with no persistent backup generally does not perform serialization.

If PDX is not enabled, Java serialization will be used. In this case, your domain classes and all nontransient properties must implement `java.io.Serializable`. This is not a requirement for PDX. Additionally, PDX is highly configurable and may be customized to optimize or enhance serialization to satisfy your application requirements.

Example 14-17 shows how to set up a GemFire repository to use PDX.

Example 14-17. Configuring a MappingPdxSerializer

```
<gfe:cache pdx-serializer="mapping-pdx-serializer" />

<bean id="mapping-pdx-serializer"
      class="org.springframework.data.gemfire.mapping.MappingPdxSerializer" />
```

The `MappingPdxSerializer` is automatically wired with the default mapping context used by the repositories. One limitation to note is that each cache instance can have only one PDX serializer, so if you're using PDX for repositories, it is advisable to set up a dedicated cache node (i.e., don't use the same process to host nonrepository regions).

Continuous Query Support

A very powerful feature of GemFire is its support for continuous queries (CQ), which provides a query-driven event notification capability. In traditional distributed applications, data consumers that depend on updates made by other processes in near-real time have to implement some type of polling scheme. This is not particularly efficient or scalable. Alternatively, using a publish-subscribe messaging system, the application, upon receiving an event, typically has to access related data stored in a disk-based data store. Continuous queries provide an extremely efficient alternative. Using CQ, the client application registers a query that is executed periodically on cache servers. The client also provides a callback that gets invoked whenever a region event affects the state of the query's result set. Note that CQ requires a client/server configuration.

Spring Data GemFire provides a `ContinuousQueryListenerContainer`, which supports a programming model based on Spring's `DefaultMessageListenerContainer` for JMS-message-driven POJOs. To configure CQ, create a CQLC using the namespace, and register a listener for each continuous query (Example 14-18). Notice that the pool must have `subscription-enabled` set to `true`, as CQ uses GemFire's subscription mechanism.

Example 14-18. Configuring a ContinuousQueryListenerContainer

```
<gfe:client-cache pool-name="client-pool" />

<gfe:pool id="client-pool" subscription-enabled="true">
  <gfe:server host="localhost" port="40404" />
</gfe:pool>

<gfe:client-region id="Person" pool-name="client-pool" />

<gfe:cq-listener-container>
  <gfe:listener ref="cqListener" query="select * from /Person" />
</gfe:cq-listener-container>

<bean id="cqListener" class="org.springframework.data.gemfire.examples.CQListener" />
```

Now, implement the listener as shown in Example 14-19.

Example 14-19. Continuous query listener implementation

```
public class CQListener {

  private static Log log = LogFactory.getLog(CQListener.class);

  public void handleEvent(CqEvent event) {
     log.info("Received event " + event);
  }
}
```

The `handleEvent()` method will be invoked whenever any process makes a change in the range of the query. Notice that `CQListener` does not need to implement any interface, nor is there anything special about the method name. The continuous query container is smart enough to automatically invoke a method that has a single `CqEvent` parameter. If there is more than one, declare the method name in the `listener` configuration.

Bibliography

[ChoDir10] Chodorow, Kristina, and Michael Dirolf. *MongoDB: The Definitive Guide*, O'Reilly Media, 2010. *http://shop.oreilly.com/product/0636920001096.do*

[CoTeGreBa11] Cogoluegnes, Arnaud, Thierry Templier, Gary Gregory, and Olivier Bazoud. *Spring Batch in Action*, Manning Publications Co., 2011. *http://www.manning.com/templier*

[Evans03] Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2003.

[Fielding00] Fielding, Roy. *Architectural Styles and the Design of Network-Based Software Architectures*, University of California, 2000. *http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm*

[Fisher12] Fisher, Mark. *Spring Integration in Action*, Manning Publications Co., 2012. *http://www.manning.com/fisher*

[Hunger12] Hunger, Michael. *Good Relationships: The Spring Data Neo4j Guide Book*, InfoQ, 2012. *http://www.infoq.com/minibooks/good-relationships-spring-data*

[Konda12] Konda, Madhusudhan. *Just Spring Data Access*, O'Reilly Media, 2012. *http://shop.oreilly.com/product/0636920025405.do*

[LongMay11] Long, Josh, and Steve Mayzak. *Getting Started with Roo*, O'Reilly Media, 2011. *http://shop.oreilly.com/product/0636920020981.do*

[Minella11] Minella, Michael T.. *Pro Spring Batch*, Apress Media LLC, 2011. *http://www.apress.com/9781430234524*

[RimPen12] Rimple, Ken, and Srini Penchikala. *Spring Roo in Action*, Manning Publications Co., 2012. *http://www.manning.com/rimple*

[WePaRo10] Webber, Jim, Savas Parastatidis, and Ian Robinson. *REST in Practice*, O'Reilly Media, 2010. *http://shop.oreilly.com/product/9780596805838.do*

Index

We'd like to hear your suggestions for improving our indexes. Send email to *index@oreilly.com*.

C

D

E

F

G

H

I

J

K

O

P

Q

R

S

T

U

V

W

X

Z

About the Authors

Dr. Mark Pollack worked on big data solutions in high-energy physics at Brookhaven National Laboratory and then moved to the financial services industry as a technical lead or architect for front-office trading systems. Always interested in best practices and improving the software development process, Mark has been a core Spring (Java) developer since 2003 and founded its Microsoft counterpart, Spring.NET, in 2004. Mark now leads the Spring Data project that aims to simplify application development with new data technologies around big data and NoSQL databases.

Oliver Gierke is an engineer at SpringSource, a division of VMware, and project lead of the Spring Data JPA, MongoDB, and core module. He has been involved in developing enterprise applications and open source projects for over six years. His working focus is centered on software architecture, Spring, and persistence technologies. He speaks regularly at German and international conferences and is the author of several technology articles.

Thomas Risberg is currently a member of the Spring Data team, focusing on the MongoDB and JDBC Extensions projects. He is also a committer on the Spring Framework project, primarily contributing to enhancements of the JDBC framework portion. Thomas works on the VMware's Cloud Foundry team, developing integration for the various frameworks and languages supported by the Cloud Foundry project. He is coauthor of *Professional Java Development with the Spring Framework*, together with Rod Johnson, Juergen Hoeller, Alef Arendsen, and Colin Sampaleanu, published by Wiley in 2005.

Jon Brisbin is a member of the SpringSource Spring Data team and focuses on providing developers with useful libraries to facilitate next-generation data manipulation. He's helped bring elements of the Grails GORM object mapper to Java-based MongoDB applications, and has provided key integration components between the Riak datastore and the RabbitMQ message broker. In addition, he blogs and speaks on evented application models, and is working diligently to bridge the gap between the bleeding-edge nonblocking and traditional JVM-based applications.

Michael Hunger has been passionate about software development for a long time. He is particularly interested in the people who develop software, software craftsmanship, programming languages, and improving code. For the last two years, he has been working with Neo Technology on the Neo4j graph database. As the project lead of Spring Data Neo4j, he helped develop the idea for a convenient and complete solution for object graph mapping. He also takes care of Neo4j cloud-hosting efforts. As a developer, Michael loves working with many aspects of programming languages, learning new things every day, participating in exciting and ambitious open source projects, and contributing to different programming-related books. Michael is also an active editor and interviewer at InfoQ.

Colophon

The animal on the cover of *Spring Data* is the giant squirrel (genus *Ratufa*), which is the largest squirrel in the world. These squirrels are found throughout tropical Asiatic forests and have a conspicuous two-toned color scheme with a distinctive white spot between the ears. Adult head and body length varies around 14 inches and the tail length is approximately 2 feet. Their ears are round and they have pronounced paws used for gripping.

A healthy adult weighs in at around four and a half pounds. With their tan, rust, brown, or beige coloring, they are possibly the most colorful of the 280 squirrel species. They are herbivorous, surviving on flowers, fruits, eggs, insects, and even bark.

The giant squirrel is an upper-canopy dwelling species, which rarely leaves the trees, and requires high branches for the construction of nests. It travels from tree to tree with jumps of up to 20 feet. When in danger, the giant squirrel often freezes or flattens itself against the tree trunk, instead of fleeing. Its main predators are birds of prey and leopards. The giant squirrel is mostly active in the early hours of the morning and in the evening, resting in the midday. It is a shy, wary animal and not easy to discover.

The cover image is from *Shaw's Zoology*. The cover font is Adobe ITC Garamond. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSansMonoCondensed.