Coherence Spring Reference Documentation

Gunnar Hillert

Version 3.0.0-SNAPSHOT

Table of Contents

Legal

Oracle licenses the Oracle Coherence Spring project under the The Universal Permissive License (UPL), Version 1.0.

The Universal Permissive License (UPL), Version 1.0

Subject to the condition set forth below, permission is hereby granted to any person obtaining a copy of this software, associated documentation and/or data (collectively the "Software"), free of charge and under any and all copyright rights in the Software, and any and all patent rights owned or freely licensable by each licensor hereunder covering either (i) the unmodified Software as contributed to or provided by such licensor, or (ii) the Larger Works (as defined below), to deal in both

(a) the Software, and (b) any piece of software and/or hardware listed in the lrgrwrks.txt file if one is included with the Software (each a "Larger Work" to which the Software is contributed by such licensors),

without restriction, including without limitation the rights to copy, create derivative works of, display, perform, and distribute the Software and make, use, sell, offer for sale, import, export, have made, and have sold the Software and the Larger Work(s), and to sublicense the foregoing rights on either these or other terms.

This license is subject to the following condition: The above copyright notice and either this complete permission notice or at a minimum a reference to the UPL must be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Chapter 1. Coherence Spring Documentation

Welcome to the reference documentation of Coherence Spring, a collection of libraries that will help you to integrate Oracle Coherence with the wider Spring ecosystem.

Oracle Coherence is a scalable, fault-tolerant, cloud-ready, distributed platform for building grid-based applications and reliably storing data. The product is used at scale, for both compute and raw storage, in a vast array of industries such as critical financial trading systems, high performance telecommunication products and e-commerce applications.

Coherence Spring features dedicated support to bootstrap Oracle Coherence and to inject Coherence resources into Spring beans as well as to inject Spring beans into Coherence resources. Spring's dependency injection (DI) support simplifies application code as Oracle Coherence *maps*, *caches* and *topics* are just injected instead of being obtained explicitly via Coherence APIs. Furthermore, using annotated *event listener* methods simplifies building reactive code that responds to Coherence cache events.

Before diving into the technical aspects of the reference documentation let's provide a brief overview of the Coherence Spring reference documentation, where to start, how to obtain further helper and more.

1.1. About the Documentation

The Coherence Spring reference guide is available as:

- Multi-page HTML
- Single page HTML
- PDF

1.2. Getting Help

If you run into issues with Spring Coherence, we are here to help.

- *Try the Quickstart.* The Quickstart will give you an overview of Coherence Spring's capabilities and provides a sample application to get you started.
- *Learn the Coherence basics.* Please have at least some basic understanding of Oracle Coherence since all Spring Coherence modules depend on it. Check out the Coherence CE web-site for general Coherence targeted reference documentation.
- *Learn the Spring basics.* The reference guide assumes that you have a basic understanding of Spring Framework and Spring Boot. Coherence Spring utilizes several other Spring projects. Check the spring.io web-site for general reference documentation. If you are starting out with Spring, try one of the guides or generate a starter project using start.spring.io/.
- *Ask a question.* Chat with us directly on Slack. We also monitor stackoverflow.com for questions tagged with oracle-coherence.
- *Contribute.* Report bugs with Spring Coherence via GitHub Issues. Both, Coherence CE and

Coherence Spring are Open Source Software (OSS) under the liberal Universal Permissive License (UPL). Contributing back is a great way to attain a deeper understanding of our projects.

All of *Coherence Spring* is open source, including the documentation. If you find problems with the docs or if you want to improve them, please get involved.

1.3. What is new?

In order to see what changes were made from earlier versions of Coherence Spring, see the Change History as well as the GitHub Releases page.

1.4. First Steps

If you are getting started with Coherence Spring, start with the Quickstart. It is a great way to see a working solution quickly. Particularly if you are relatively new to Spring, continue with the Coherence Spring Boot chapter next. The reference documentation makes a distinction between Spring Framework and Spring Boot. At its very core, Spring Framework provides Dependency Injection (DI) or Inversion Of Control (IOC) to Java applications. Furthermore, Spring Framework gives developers comprehensive infrastructure support for developing Java applications.

Spring Boot on the other hand, is an opinionated extension to the Spring Framework by:

- Eliminating boilerplate configurations
- Providing Auto-Configuration for other Spring modules and third-party integrations
- Metrics + health checks

The vast majority of new Spring projects will utilize Spring Boot. Nonetheless, please also study the Spring Framework targeted chapters as Spring Frameworks is the foundation for everything related to Spring Boot.

Chapter 2. Quickstart

In this getting started chapter we will look a demo to illustrate basic usage of Oracle Coherence when using it with Spring. This demo provides an example of using Coherence Spring's Cache Abstraction.

The demo application is basically a super-simple event manager. We can create `Events` and assign `People` to them using an exposed REST API. The data is saved in an embedded HSQL database. The caching is implemented at the service layer:

When an `Event` is created, it is not only persisted to the database but also *put* to the Coherence Cache. Therefore, whenever an `Event` is retrieved, it will be returned from the Coherence Cache. You can also delete `Events`, in which case the `Event` will be *evicted* from the cache. You can perform the same CRUD operations for people as well.

2.1. How to Run the Demo

In order to get started, please checkout the code from the coherence-community/coherence-spring[Coherence Spring Repository] GitHub repository.

Clone GitHub Repository

```
$ git clone https://github.com/coherence-community/coherence-spring.git
$ cd coherence-spring
```

You now have checked out all the code for Coherence Spring. The relevant demo code for this Quickstart demo is under `coherence-spring-samples/coherence-spring-demo/`.

There you will find 3 Maven sub-modules:

- coherence-spring-demo-classic
- coherence-spring-demo-boot
- coherence-spring-demo-core

The first two Maven modules are essentially variations of the same app. The third module contains shared code.

coherence-spring-demo-classic	Provides a demo using **Spring Framework** without Spring Boot
coherence-spring-demo-boot	Provides a demo using **Spring Boot**
coherence-spring-demo-core	Contains common code shared between the two apps

In this chapter we will focus on the **Spring Boot** version. Since we checked out the project, let's build it using Maven:

Build the project

```
 $ ./mvnw clean package -pl coherence-spring-samples/coherence-spring-demo/coherence-spring-demo-boot
```

Now we are ready to run the application:

Run the Spring Boot application

```
 $ java -jar coherence-spring-samples/coherence-spring-demo/coherence-spring-demo-boot/target/coherence-spring-demo-boot-3.0.0-SNAPSHOT.jar
```

2.2. Interacting with the Cache

Once the application is started, the embedded database is empty. Let's create an event with 2 people added to them using curl:

Create the first event

```
 curl --request POST 'http://localhost:8080/api/events?title=First%20Event&date=2020-11-30'
```

This call will create and persist an `Event` to the database. However, there is more going on. The created `Event` is also added to the Coherence Cache. The magic is happening in the Service layer, specifically in `DefaultEventService#createAndStoreEvent()`, which is annotated with `@CachePut(cacheNames="events", key="#result.id")`.

The `cacheNames` attribute of the `@CachePut` annotation indicates the name of the underlying cache to use. As caches are basically just a Map, we also need a key. In this case we use the expression `#result.id` to retrieve the primary key of the `Event` as it was persisted. Thus, the saved `Event` is added to the cache named `events` and ultimately also returned and printed to the console:

Return result of the created event

```
{
  "id" : 1,
  "title" : "First Event",
  "date" : "2020-11-30T00:00:00.000+00:00"
}
```

We see that an Event with the id `1` was successfully created. Let's verify that the *cache put* worked by looking at the cache statistics:

Retrieving Cache Statistics

```
$ curl --request GET 'http://localhost:8080/api/statistics/events'
```

In the console you should see some basic statistics being printed including `totalPuts : 1`:

Cache Statistic Results

```
{
  "averageMissMillis" : 0.0,
  "cachePrunesMillis" : 0,
  "averagePruneMillis" : 0.0,
  "totalGetsMillis" : 0,
  "averageGetMillis" : 0.0,
  "totalPutsMillis" : 11,
  "averagePutMillis" : 11.0,
  "cacheHitsMillis" : 0,
  "averageHitMillis" : 0.0,
  "cacheMissesMillis" : 0,
  "cacheHits" : 0,
  "cacheMisses" : 0,
  "hitProbability" : 0.0,
  "totalPuts" : 1,
  "totalGets" : 0,
  "cachePrunes" : 0
}
```

Next, lets retrieve the Event using id 1:

Retrieve Event

```
curl --request GET 'http://localhost:8080/api/events/1'
```

The Event is returned. Did you notice? No SQL queries were executed as the value was directly retrieved from the Cache. Let's check the statistics again by executing:

Retrieve Cache Statistics

```
curl --request GET 'http://localhost:8080/api/statistics/events'
```

We will see now how values are being returned from the cache by seeing increasing cacheHits, e.g. `"cacheHits" : 1`. Let's evict our Event with id 1 from the cache named events:

Evict Event

```
curl --request DELETE 'http://localhost:8080/api/events/1'
```

If you now retrieve the event again using:

Retrieve Event

```
curl --request GET 'http://localhost:8080/api/events/1'
```

you will see an SQL query executed in the console, re-populating the cache. Feel free to play along with the Rest API. We can for example add people:

Add people

```
 curl --request POST
'http://localhost:8080/api/people?firstName=Conrad&lastName=Zuse&age=85'
 curl --request POST
'http://localhost:8080/api/people?firstName=Alan&lastName=Turing&age=41'
```

List people

```
curl --request GET 'http://localhost:8080/api/people'
```

Or assign people to events:

Assign People to Events

```
curl --request POST 'http://localhost:8080/api/people/2/add-to-event/1'
curl --request POST 'http://localhost:8080/api/people/3/add-to-event/1'
```

2.3. Behind the Scenes

What is involved to make this all work? Using Spring Boot, the setup is incredibly simple. We take advantage of Spring Boot's AutoConfiguration capabilities, and the sensible defaults provided by *Coherence Spring*.

In order to activate AutoConfiguration for Coherence Spring you need to add the `coherence-spring-boot-starter` dependency as well as the desired dependency for Coherence.

POM configuration

```
<dependency>
    <groupId>com.oracle.coherence.spring</groupId>
    <artifactId>coherence-spring-boot-starter</artifactId> ①
    <version>3.0.0-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>com.oracle.coherence.ce</groupId>
    <artifactId>coherence</artifactId>                     ②
    <version>21.06-M1</version>
</dependency>
```

① Activate Autoconfiguration by adding the `coherence-spring-boot-starter` dependency

② Add the desired version of Coherence (CE or Commercial)

In this quickstart example we are using Spring's Caching abstraction and therefore, we use the `spring-boot-starter-cache` dependency as well:

POM configuration for Spring Cache Abstraction

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-cache</artifactId>
</dependency>
```

For caching you also must activate caching using the `@EnableCaching` annotation.

Spring Boot App configuration

```
@SpringBootApplication
@EnableCaching                                          ①
public class CoherenceSpringBootDemoApplication {

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

}
```

① Activate the Spring Cache Abstraction

Please see the relevant chapter on Caching in the Spring Boot reference guide.

With `@EnableCaching` in place, Coherence's autoconfiguration will also provide a `CoherenceCacheManager` bean to the application context.

Chapter 3. Coherence Spring Core

This section dives into the Coherence Spring Core module. Coherence Spring Core provides the basic support for the Spring Framework.

3.1. Getting Started

To add support for Oracle Coherence to an existing Spring Framework project, you should first add the required Spring Coherence dependencies to your build configuration:

Example 1. Coherence Spring Dependencies

Maven

```
<dependency>
    <groupId>com.oracle.coherence.spring</groupId>
    <artifactId>coherence-spring-core</artifactId>
    <version>3.0.0-SNAPSHOT</version>
</dependency>
```

Gradle

```
implementation("com.oracle.coherence.spring:coherence-spring-core:3.0.0-SNAPSHOT")
```

Next you also need to add the version of Coherence that your application will be using. Coherence Spring is compatible with both the open source Coherence CE and the commercial version of Oracle Coherence. Therefore, we don't bring in Oracle Coherence as transitive dependency. For example, to use Coherence CE specify:

Example 2. Oracle Coherence CE Dependency

Maven

```
<dependency>
    <groupId>com.oracle.coherence.ce</groupId>
    <artifactId>coherence</artifactId>
    <version>21.06-M1</version>
</dependency>
```

Gradle

```
implementation("com.oracle.coherence.ce:coherence:21.06-M1")
```

In order to use the commercial version of Coherence:

Example 3. Commercial Oracle Coherence Dependency

Maven

```
<dependency>
    <groupId>com.oracle.coherence</groupId>
    <artifactId>coherence</artifactId>
    <version>14.1.1-0-1</version>
</dependency>
```

Gradle

```
implementation("com.oracle.coherence.ce:coherence:14.1.1-0-1")
```

Coherence CE versions are available from Maven Central. The commercial versions of Coherence needs to be uploaded into your own Maven repository.

3.2. Bootstrapping Coherence

The default behaviour of Coherence Spring is to use the Coherence bootstrap API introduced in Coherence CE `20.12` to configure and create Coherence instances. This means that Coherence resources in a Spring application are typically part of a Coherence Session.

By default, Coherence will start a single Session configured to use the default Coherence configuration file. This behavior can easily be configured using traditional Coherence using system properties or using dedicated configuration.

3.3. Using the Default Session

The main building block for setting up Coherence for Spring is the `@EnableCoherence` annotation. This annotation will import the `CoherenceSpringConfiguration` under the covers. Therefore, you can alternatively also declare `@Import(CoherenceSpringConfiguration.class)` instead. Without providing any further configuration the default session is configured using the embedded default configuration file.

To provide further customization, you may for example provide an implementation of the `AbstractSessionConfigurationBean`, e.g.:

SessionConfigurationBean

```
@Bean
SessionConfigurationBean sessionConfigurationBeanDefault() {
    final SessionConfigurationBean sessionConfigurationBean =
            new SessionConfigurationBean();
    sessionConfigurationBean.setType(SessionType.SERVER);
    sessionConfigurationBean.setConfig("test-coherence-config.xml");
    return sessionConfigurationBean;
}
```

3.4. Configure Multiple Sessions

3.5. Session Configuration Properties

3.6. Events

Event driven patterns are a common way to build scalable applications and microservices. Coherence produces a number of events that can be used by applications to respond to data changes and other actions in Coherence.

There are two types of events in Coherence:

- MapEvents which are subscribed to using a MapListener
- Events, which are subscribed to using an EventInterceptor

Spring makes subscribing to both of these event-types much simpler using observer methods annotated with @CoherenceEventListener.

Example of using a Coherence Event Listener

```
@CoherenceEventListener
void onEvent(CoherenceLifecycleEvent event) {
    // TODO: process event...
}
```

The method above receives all events of type `CoherenceLifecycleEvent` emitted during the lifetime of the application. The actual events received can be controlled further by annotating the method or the method arguments.

Spring `4.2` introduced Annotation-driven event listeners as part of its event support. Coherence Spring does **NOT** directly use Spring's `ApplicationEvent` class and the corresponding `ApplicationListener` interface. However, Coherence Spring follows that pattern conceptually in order to provide a similar user experience.

3.6.1. MapEvent Listeners

Listening for changes to data in Coherence is a common use case in applications. Typically, this involves creating an implementation of a MapListener and adding that listener to a NamedMap or NamedCache. Using Coherence Spring makes this much simpler by just using Spring beans with suitably annotated observer methods that will receive the respective events.

3.6.1.1. MapEvent Observer Methods

A MapEvent observer method is a method on a Spring bean that is annotated with @CoherenceEventListener. The annotated method must have a void return type and must take a single method parameter of type MapEvent, typically this has the generic types of the underlying map/cache key and value.

For example, assuming that there is a map/cache named people, with keys of type String and values of type Plant, and the application has logic that should be executed each time a new Plant is inserted into the map:

Example of listening to Inserted events

```
import com.oracle.coherence.spring.annotation.event.Inserted;
import com.oracle.coherence.spring.annotation.event.MapName;
import com.oracle.coherence.spring.event.CoherenceEventListener;
import com.tangosol.util.MapEvent;
import org.springframework.stereotype.Component;

@Component                                                    ①
public class PersonEventHandler {

    @CoherenceEventListener                                   ②
    public void onNewPerson(@MapName("people")                ③
                            @Inserted                         ④
                            MapEvent<String, Person> event) {
        // TODO: process the event
    }
}
```

① The PersonController is a simple Spring bean, in this case a Controller.

② The onNewPerson method is annotated with @CoherenceEventListener making it a Coherence event listener.

③ The @MapName("people") annotation specifies the name of the map to receive events from, in this case people.

④ The @Inserted annotation specified that only Inserted events should be sent to this method.

The above example is still rather simple. There are a number of other annotations that provide much finer-grained control over what events are received from where.

Specify the Map/Cache name

By default, a `MapEvent` observer method would receive events for all maps/caches. In practice though, this would not be a very common use case, and typically you would want an observer method to listen to events that are for specific caches. The Coherence Spring API contains two annotations for specifying the map name:

- @MapName
- @CacheName

Both annotations take a single `String` value that represents the name of the map or cache that events should be received from.

Listening to events for all caches

```
    @CoherenceEventListener
    public void onEvent(MapEvent<String, String> event) {
        // TODO: process the event
    }
```

The above method receives events for *all* caches.

Listening to events for the map named "foo"

```
    @CoherenceEventListener
    public void onFooEvent(@MapName("foo")                ①
                           MapEvent<String, String> event) {
        // TODO: process the event
    }
```

① The above method receives events for the map named `foo`.

Listening to events for the cache named "bar"

```
    @CoherenceEventListener
    public void onBarEvent(@CacheName("bar")              ①
                           MapEvent<String, String> event) {
        // TODO: process the event
    }
```

① The above method receives events for the cache named `bar`.

Specify the Cache Service name

In the previous section we showed to restrict received events to a specific map or cache name. Events can also be restricted to only events from a specific cache service. In Coherence all caches are owned by a cache service, which has a unique name. By default, a `MapEvent` observer method would receive events for a matching cache name on *all* services. If an applications Coherence configuration has multiple services, the events can be restricted to just specific services using the @ServiceName annotation.

Listening to events for the "foo" map on all services

```
@CoherenceEventListener
public void onEventFromAllServices(@MapName("foo")  ①
                           MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for the map named `foo` on *all* cache services.

Listening to events for the "foo" map on the "Storage" service only

```
@CoherenceEventListener
public void onEventOnStorageService(@MapName("foo")
                    @ServiceName("Storage")        ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for the map named `foo` owned by the cache service named `Storage`.

Listening to events for ALL caches on the "Storage" service

```
@CoherenceEventListener
public void onEventFromAllCachesOnStorageService(@ServiceName("Storage")  ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for *all* caches owned by the cache service named `Storage` as there is no `@MapName` or `@CacheName` annotation.

Specify the Owning Session Name

In applications that use multiple `Sessions`, there may be a situation where more than one session has a map with the same name. In those cases an observer method may need to restrict the events it receives to a specific session. The events can be restricted to `maps` and/or `caches` in specific sessions using the @SessionName annotation.

Listening to events for the "orders" map in ALL sessions

```
@CoherenceEventListener
public void onOrdersEventAllSessions(@MapName("orders")                ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for the map named `orders` in *all* sessions.

Listening to events for the "orders" map in the "Customer" session only

```
@CoherenceEventListener
public void onOrdersEventInCustomerSession(@MapName("orders")
                    @SessionName("Customer")                          ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for the map named `orders` owned by the `Session` named `Customer`.

Listening to events for ALL caches in the "Customer" session

```
@CoherenceEventListener
public void onEventInAllCachesInCustomerSession(@SessionName("Customer") ①
                           MapEvent<String, String> event) {
    // TODO: process the event
}
```

① The above method receives events for the *all* caches owned by the `Session` named `Customer` as there is no `@MapName` or `@CacheName` annotation.

Therefore, in application with multiple sessions, events with the same name can be routed by session.

Route events with the cache name by the name of the session

```
@CoherenceEventListener
public void onCustomerOrders(@SessionName("Customer")                 ①
                             @MapName("orders")
                             MapEvent<String, Order> event) {
    // TODO: process the event
}

@CoherenceEventListener
public void onCatalogOrders(@SessionName("Catalog")                   ②
                            @MapName("orders")
                            MapEvent<String, Order> event) {
    // TODO: process the event
}
```

① The `onCustomerOrders` method will receive events for the `orders` map owned by the `Session` named `Customer`.

② The `onCatalogOrders` method will receive events for the `orders` map owned by the `Session` named `Catalog`.

3.6.1.2. Receive Specific Event Types

There are three types of event that a `MapEvent` observer method can receive:

- `Insert`
- `Update`
- `Delete`

By default, an observer method will receive all events for the map (or maps) it applies to. This can be controlled using the following annotations:

- `@Inserted` - to receive `Insert` events.
- `@Updated` - to receive `Update` events.
- `@Deleted` - to receive `Delete` events.

Zero or more of the above annotations can be used to annotate the `MapEvent` parameter of the observer method.

Listen to "Insert" event for the "test" map only

```
@CoherenceEventListener
public void onInsertEvent(@MapName("test")
                    @Inserted                                          ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① Only `Insert` events for the map `test` will be received.

Listen to "Insert" and "Delete" events for the "test" map only

```
@CoherenceEventListener
public void onInsertAndDeleteEvent(@MapName("test")
                    @Inserted @Deleted                                 ①
                    MapEvent<String, String> event) {
    // TODO: process the event
}
```

① Only `Insert` and `Delete` events for the map `test` will be received.

Listen to ALL map events for the "test" map

```
@CoherenceEventListener
public void onMapEvent(@MapName("test") MapEvent<String, String> event) {
    // TODO: process the event
}
```

All events for the map `test` will be received.

3.6.1.3. Filtering Events

The `MapEvents` received by an observer method can be further restricted by applying a filter. Filters are applied by annotating the method with a filter binding annotation, which is a link to a factory

that creates a specific instance of a Filter. Event filters applied in this way are executed on the server, which can make receiving events more efficient for clients, as the event will not be sent from the server at all.

Coherence Spring comes with some built in implementations, for example:

- @AlwaysFilter,
- @WhereFilter,

It is simple to implement custom filters as required by applications. Please refer to the Filter Binding Annotation section for more details.

For example, let's assume there is a map named `people` with keys of type `String` and values of type `People`, and an observer method needs to receive events for all values where the `age` property is `18` or over. A custom filter binding annotation could be written to create the required `Filter`. However, as the condition is very simple, the built-in `@WhereFilter` filter binding annotation will be used in this example with a where-clause of `age >= 18`.

Example of a Where Filter

```
@WhereFilter("age >= 18")                                          ①
@CoherenceEventListener
@MapName("people")
public void onAdult(MapEvent<String, Person> people) {
    // TODO: process event...
}
```

① The `@WhereFilter` annotation is applied to the method.

The `onAdult` method above will receive all events emitted from the `people` map, but only for entries where the value of the `age` property of the entry value is `>= 18`.

3.6.1.4. Transforming Events

In some use-cases the `MapEvent` observer method does not require the whole map or cache value to process, it might only require one, or a few, properties of the value, or it might require some calculated value. This can be achieved by using an event transformer to convert the values that will be received by the observer method. The transformation takes place on the server before the event is emitted to the method. This can improve efficiency on a client in cases where the cache value is large, but the client only requires a small part of that value because only the required values are sent over the wire to the client.

In Coherence Spring, event values are transformed using a ValueExtractor. A `ValueExtractor` is a simple interface that takes in one value and transforms it into another value. The `ValueExtractor` is applied to the event value. As events contain both a new and old values, the extractor is applied to both as applicable. For `Insert` events there is only a new value, for `Update` events there will be both, a new and an old value, and for `Delete` events, there will only be an old value. The extractor is not applied to the event key.

The `ValueExtractor` to use for a `MapEvent` observer method is indicated by annotating the method

with an extractor binding annotation. An extractor binding is an annotation that is itself annotated with the meta-annotation @ExtractorBinding. The extractor binding annotation is a link to a corresponding ExtractorFactory that will build an instance of a ValueExtractor.

For example, assuming that there is a NamedMap with the name orders that has keys of type String and values of type Order. The Order class has a customerId property of type String. A MapEvent observer method is only interested in the customerId for an order, so the built-in extractor binding annotation @PropertyExtractor can be used to just extract the customerId from the event:

Example of a Property Extractor

```
@CoherenceEventListener
@PropertyExtractor("customerId")                                    ①
public void onOrder(@MapName("orders")                              ②
                    MapEvent<String, String> event) {               ③
    // TODO: process event...
}
```

① The method is annotated with @PropertyExtractor to indicate that a ValueExtractor that just extracts the customerId property should be used to transform the event.

② The map name to receive events from is set to orders

③ Note that the generic types of the MapEvent parameter are now MapEvent<String, String> instead of MapEvent<String, Order> because the event values will have been transformed from an Order into just the String customerId.

It is possible to apply multiple filter binding annotations to a method. In this case the extractors are combined into a Coherence ChainedExtractor, which will return the extracted values as a java.util.List.

Expanding on the example above, if the Order class also has an orderId property of type Long, and an observer method, only interested in Insert events needs both the customerId and orderId, then the method can be annotated with a two @PropertyExtractor annotations:

Example of using multiple Property Extractors

```
@CoherenceEventListener
@PropertyExtractor("customerId")                                    ①
@PropertyExtractor("orderId")
public void onOrderWithMultiplePropertyExtractors(
                    @Inserted                                       ②
                    @MapName("orders")
                    MapEvent<String, List<Object>> event) {         ③
    List list = event.getNewValue();
    String customerId = (String) list.get(0);                       ④
    Long orderId = (Long) list.get(1);
    // ...
}
```

① The method is annotated with two @PropertyExtractor annotations, one to extract customerId and

one to extract `orderId`.

② The method parameter is annotated with `@Inserted` so that the method only receives `Insert` events.

③ The `MapEvent` parameter not has a key of type `String` and a value of type `List<Object>`, because the values from the multiple extractors will be returned in a `List`. We cannot use a generic value narrower than `Object` for the list because it will contain a `String` and a `Long`.

④ The extracted values can be obtained from the list, they will be in the same order that the annotations were applied to the method.

3.6.2. Coherence Event Interceptors

Coherence produces many events in response to various server-side and client-side actions. For example, *Lifecycle events* for Coherence itself, maps and cache, *Entry events* when data in maps and caches changes, *Partition events* for partition lifecycle and distribution, *EntryProcessor events* when invoked on a map or cache, etc. In a stand-alone Coherence application these events are subscribed to using a EventInterceptor implementation registered to listen to specific event types.

The Coherence Spring API makes subscribing to these events simple, by using the same approach used for Spring Application events, namely annotated event observer methods. A Coherence event observer method is a method annotated with @CoherenceEventListener that has a `void` return type, and a single parameter of the type of event to be received. The exact events received can be further controlled by applying other annotations to the method or event parameter. The annotations applied will vary depending on the type of the event.

3.6.2.1. Event Types

The different types of event that can be observed are listed below:

- CoherenceLifecycleEvent - lifecycle events for Coherence instances
- SessionLifecycleEvent - lifecycle events for Session instances
- LifecycleEvent - lifecycle events for ConfigurableCacheFactory instances
- CacheLifecycleEvent - lifecycle events for NamedMap and NamedCache instances
- EntryEvent - events emitted by the mutation of entries in a NamedMap or NamedCache
- EntryProcessorEvent - events emitted by the invocation of an EntryProcessor on entries in a NamedMap or NamedCache
- TransactionEvent - events pertaining to all mutations performed within the context of a single request in a partition of a NamedMap or NamedCache, also referred to as "partition level transactions".
- TransferEvent - captures information concerning the transfer of a partition for a storage enabled member.
- UnsolicitedCommitEvent - captures changes pertaining to all observed mutations performed against caches that were not directly caused (solicited) by the partitioned service. These events may be due to changes made internally by the backing map, such as eviction, or referrers of the backing map causing changes.

- If using commercial versions of Coherence with Coherence Spring, there are also events associated to the federation of data between different clusters.

Most of the events above only apply to storage enabled cluster members. For example, an `EntryEvent` will only be emitted for mutations of an entry on the storage enabled cluster member that owns that entry. Lifecycle events on the other hand, may be emitted on all members, such as `CacheLifecycle` event that may be emitted on any member when a cache is created, truncated, or destroyed.

3.6.2.2. Coherence Lifecycle Events

LifecycleEvent are emitted to indicate the lifecycle of a ConfigurableCacheFactory instance.

To subscribe to `LifecycleEvent` simply create a Spring bean with a listener method that is annotated with @CoherenceEventListener. The method should have a single parameter of type `LifecycleEvent`.

`LifecycleEvent` are emitted by `ConfigurableCacheFactory` instances and will only be received in the same JVM, which could be a cluster member or a client.

For example, the `onEvent` method below will receive lifecycle events for all `ConfigurableCacheFactory` instances in the current application:

```
@CoherenceEventListener
public void onEvent(LifecycleEvent event) {
    // TODO: process the event
}
```

Receive Specific LifecycleEvent Types

There are four different types of `LifecycleEvent`. By adding the corresponding annotation to the method parameter the method will only receive the specified events.

- **Activating** - a `ConfigurableCacheFactory` instance is about to be activated, use the @Activating annotation
- **Activated** - a `ConfigurableCacheFactory` instance has been activated, use the @Activated annotation
- **Disposing** - a `ConfigurableCacheFactory` instance is about to be disposed, use the @Disposing annotation

For example, the method below will only receive `Activated` and `Disposing` events.

```
@CoherenceEventListener
public void onEvent(@Activated @Disposing LifecycleEvent event) {
    // TODO: process the event
}
```

Receive CoherenceLifecycleEvents for a Specific Coherence Instance

Each `Coherence` instance in an application has a unique name. The observer method can be annotated to only receive events associated with a specific `Coherence` instance by using the `@Name` annotation.

For example, the method below will only receive events for the `Coherence` instance named `customers`:

```
@CoherenceEventListener
public void onEvent(@Name("customers") CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

The method in this example will receive events for the default `Coherence` instance:

```
@CoherenceEventListener
public void onEvent(@Name(Coherence.DEFAULT_NAME) CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

3.6.2.3. Session Lifecycle Events

SessionLifecycleEvents are emitted to indicate the lifecycle event of a Session instance.

To subscribe to `SessionLifecycleEvents` simply create a Spring bean with a listener method annotated with `@CoherenceEventListener`. The method should have a single parameter of type `SessionLifecycleEvent`.

`SessionLifecycleEvents` are emitted by `Session` instances and will only be received in the same JVM, which could be a cluster member or a client.

For example, the `onEvent` method below will receive lifecycle events for all `Session` instances in the current application:

```
@CoherenceEventListener
public void onEvent(SessionLifecycleEvent event) {
    // TODO: process the event
}
```

Receive Specific SessionLifecycleEvent Types

There are four different types of `SessionLifecycleEvent`. By adding the corresponding annotation to the method parameter the method will only receive the specified events.

- **Starting** - a `Coherence` instance is about to start, use the `@Starting` annotation
- **Started** - a `Coherence` instance has started, use the `@Started` annotation

- **Stopping** - a `Coherence` instance is about to stop, use the @Stopping annotation
- **Stopped** - a `Coherence` instance has stopped, use the @Stopped annotation

For example, the method below will only receive `Started` and `Stopped` events.

```
@CoherenceEventListener
public void onEvent(@Started @Stopped SessionLifecycleEvent event) {
    // TODO: process the event
}
```

Receive SessionLifecycleEvents for a Specific Session Instance

Each `Session` instance in an application has a name. The observer method can be annotated to only receive events associated with a specific `Session` instance by using the @Name annotation.

For example, the method below will only receive events for the `Session` instance named `customers`:

```
@CoherenceEventListener
public void onEvent(@Name("customers") SessionLifecycleEvent event) {
    // TODO: process the event
}
```

The method in this example will receive events for the default `Coherence` instance:

```
@CoherenceEventListener
public void onEvent(@Name(Coherence.DEFAULT_NAME) SessionLifecycleEvent event) {
    // TODO: process the event
}
```

3.6.2.4. ConfigurableCacheFactory Lifecycle Events

CoherenceLifecycleEvents are emitted to indicate the lifecycle of a Coherence instance.

To subscribe to `CoherenceLifecycleEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `CoherenceLifecycleEvent`.

`CoherenceLifecycleEvent` are emitted by `Coherence` instances and will only be received in the same JVM, which could be a cluster member or a client.

For example, the `onEvent` method below will receive lifecycle events for all `Coherence` instances in the current application:

```
@CoherenceEventListener
public void onEvent(CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

Receive Specific CoherenceLifecycleEvent Types

There are four different types of `CoherenceLifecycleEvent`. By adding the corresponding annotation to the method parameter the method will only receive the specified events.

- **Starting** - a `Coherence` instance is about to start, use the `@Starting` annotation
- **Started** - a `Coherence` instance has started, use the `@Started` annotation
- **Stopping** - a `Coherence` instance is about to stop, use the `@Stopping` annotation
- **Stopped** - a `Coherence` instance has stopped, use the `@Stopped` annotation

For example, the method below will only receive `Started` and `Stopped` events.

```
@CoherenceEventListener
public void onEvent(@Started @Stopped CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

Receive CoherenceLifecycleEvents for a Specific Coherence Instance

Each `Coherence` instance in an application has a unique name. The observer method can be annotated to only receive events associated with a specific `Coherence` instance by using the `@Name` annotation.

For example, the method below will only receive events for the `Coherence` instance named `customers`:

```
@CoherenceEventListener
public void onEvent(@Name("customers") CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

The method in this example will receive events for the default `Coherence` instance:

```
@CoherenceEventListener
public void onEvent(@Name(Coherence.DEFAULT_NAME) CoherenceLifecycleEvent event) {
    // TODO: process the event
}
```

3.6.2.5. Cache Lifecycle Events

CacheLifecycleEvent are emitted to indicate the lifecycle of a cache instance.

To subscribe to `CacheLifecycleEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `CacheLifecycleEvent`.

For example, the `onEvent` method below will receive lifecycle events for all caches.

```
@CoherenceEventListener
public void onEvent(CacheLifecycleEvent event) {
    // TODO: process the event
}
```

Receive Specific CacheLifecycleEvent Types

There are three types of `CacheLifecycleEvent:

- **Created** - a cache instance has been created, use the @Created annotation
- **Truncated** - a cache instance has been truncated (all data was removed), use the @Truncated annotation
- **Destroyed** - a cache has been destroyed (destroy is a cluster wide operation, so the cache is destroyed on all members of the cluster and clients) use the @Destroyed annotation

For example, the method below will only receive `Created` and `Destroyed` events for all caches.

```
@CoherenceEventListener
public void onEvent(@Created @Destroyed CacheLifecycleEvent event) {
    // TODO: process the event
}
```

Receive CacheLifecycleEvents for a Specific NamedMap or NamedCache

To only receive events for a specific `NamedMap` annotate the method parameter with the @MapName annotation. To only receive events for a specific `NamedCache` annotate the method parameter with the @CacheName annotation.

The `@MapName` and `@CacheName` annotations are actually interchangeable so use whichever reads better for your application code, i.e. if your code is dealing with `NamedMap` used `@MapName`. At the storage level, where the events are generated a `NamedMap` and `NamedCache` are the same.

The method below will only receive events for the map named `orders`:

```
@CoherenceEventListener
public void onEvent(@MapName("orders") CacheLifecycleEvent event) {
    // TODO: process the event
}
```

Receive CacheLifecycleEvents from a Specific Cache Service

Caches are owned by a Cache Service, it is possible to restrict events received by a method to only those related to caches owned by a specific service by annotating the method parameter with the @ServiceName annotation.

The method below will only receive events for the caches owned by the service named `StorageService`:

```
@CoherenceEventListener
public void onEvent(@ServiceName("StorageService") CacheLifecycleEvent event) {
    // TODO: process the event
}
```

Receive CacheLifecycleEvents from a Specific Session

A typical use case is to obtain `NamedCache` and `NamedMap` instances from a `Session`. It is possible to restrict events received by a method to only those related to caches owned by a specific `Session` by annotating the method parameter with the @SessionName annotation.

The method below will only receive events for the caches owned by the `Session` named `BackEnd`:

```
@CoherenceEventListener
public void onEvent(@SessionName("BackEnd") CacheLifecycleEvent event) {
    // TODO: process the event
}
```

3.6.2.6. Entry Events

An EntryProcessorEvent is emitted when a EntryProcessor is invoked on a cache. These events are only emitted on the storage enabled member that is the primary owner of the entry that the `EntryProcessor` is invoked on.

To subscribe to `EntryProcessorEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `EntryProcessorEvent`.

For example, the `onEvent` method below will receive entry events for all caches.

```
@CoherenceEventListener
public void onEvent(EntryProcessorEvent event) {
    // TODO: process the event
}
```

Receive Specific EntryProcessorEvent Types

There are a number of different `EntryProcessorEvent` types.

- **Inserting** - an entry is being inserted into a cache, use the @Inserting annotation
- **Inserted** - an entry has been inserted into a cache, use the @Inserted annotation
- **Updating** - an entry is being updated in a cache, use the @Updating annotation
- **Updated** - an entry has been updated in a cache, use the @Updated annotation
- **Deleting** - an entry is being deleted from a cache, use the @Deleting annotation
- **Deleted** - an entry has been deleted from a cache, use the @Deleted annotation

To restrict the `EntryProcessorEvent` types received by a method apply one or more of the annotations above to the method parameter. For example, the method below will receive `Inserted` and `Deleted` events.

```
@CoherenceEventListener
public void onEvent(@Inserted @Deleted EntryProcessorEvent event) {
    // TODO: process the event
}
```

The event types fall into two categories, pre-events (those name *ing) and post-events, those named *ed). Pre-events are emitted synchronously before the entry is mutated. Post-events are emitted asynchronously after the entry has been mutated.

As pre-events are synchronous the listener method should not take a long time to execute as it is blocking the cache mutation and could obviously be a performance impact. It is also important that developers understand Coherence reentrancy as the pre-events are executing on the Cache Service thread so cannot call into caches owned by the same service.

Receive EntryProcessorEvents for a Specific NamedMap or NamedCache

To only receive events for a specific `NamedMap` annotate the method parameter with the @MapName annotation. To only receive events for a specific `NamedCache` annotate the method parameter with the @CacheName annotation.

The `@MapName` and `@CacheName` annotations are actually interchangeable so use whichever reads better for your application code, i.e. if your code is dealing with `NamedMap` used `@MapName`. At the storage level, where the events are generated a `NamedMap` and `NamedCache` are the same.

The method below will only receive events for the map named `orders`:

```
@CoherenceEventListener
public void onEvent(@MapName("orders") EntryProcessorEvent event) {
    // TODO: process the event
}
```

Receive EntryProcessorEvents from a Specific Cache Service

Caches are owned by a Cache Service, it is possible to restrict events received by a method to only those related to caches owned by a specific service by annotating the method parameter with the @ServiceName annotation.

The method below will only receive events for the caches owned by the service named `StorageService`:

```
@CoherenceEventListener
public void onEvent(@ServiceName("StorageService") EntryProcessorEvents event) {
    // TODO: process the event
}
```

Receive EntryProcessorEvents from a Specific Session

A typical use case is to obtain `NamedCache` and `NamedMap` instances from a `Session`. It is possible to restrict events received by a method to only those related to caches owned by a specific `Session` by annotating the method parameter with the @SessionName annotation.

The method below will only receive events for the caches owned by the `Session` named `BackEnd`:

```
@CoherenceEventListener
public void onEvent(@SessionName("BackEnd") EntryProcessorEvents event) {
    // TODO: process the event
}
```

3.6.2.7. EntryProcessor Events

An EntryProcessorEvent is emitted when a mutation occurs on an entry in a cache. These events are only emitted on the storage enabled member that is the primary owner of the entry.

To subscribe to `EntryProcessorEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `EntryProcessorEvent`.

For example, the `onEvent` method below will receive entry events for all caches.

```
@CoherenceEventListener
public void onEvent(EntryProcessorEvent event) {
    // TODO: process the event
}
```

Receive Specific EntryProcessorEvent Types

There are a number of different `EntryProcessorEvent` types.

- Executing - an `EntryProcessor` is being invoked on a cache, use the `@Executing` annotation
- Executed - an `EntryProcessor` has been invoked on a cache, use the `@Executed` annotation

To restrict the `EntryProcessorEvent` types received by a method apply one or more of the annotations above to the method parameter. For example, the method below will receive `Executed` events.

```
@CoherenceEventListener
public void onEvent(@Executed EntryProcessorEvent event) {
    // TODO: process the event
}
```

The event types fall into two categories, pre-event ('Executing') and post-event (`Executed`). Pre-events are emitted synchronously before the `EntryProcessor` is invoked. Post-events are emitted asynchronously after the `EntryProcessor` has been invoked.

As pre-events are synchronous the listener method should not take a long time to execute as it is blocking the `EntryProcessor` invocation and could obviously be a performance impact. It is also important that developers understand Coherence reentrancy as the pre-events are executing on the Cache Service thread so cannot call into caches owned by the same service.

Receive EntryProcessorEvents for a Specific NamedMap or NamedCache

To only receive events for a specific `NamedMap` annotate the method parameter with the `@MapName` annotation. To only receive events for a specific `NamedCache` annotate the method parameter with the `@CacheName` annotation.

The `@MapName` and `@CacheName` annotations are actually interchangeable so use whichever reads better for your application code, i.e. if your code is dealing with `NamedMap` used `@MapName`. At the storage level, where the events are generated a `NamedMap` and `NamedCache` are the same.

The method below will only receive events for the map named `orders`:

```
@CoherenceEventListener
public void onEvent(@MapName("orders") EntryProcessorEvent event) {
    // TODO: process the event
}
```

Receive EntryProcessorEvents from a Specific Cache Service

Caches are owned by a Cache Service, it is possible to restrict events received by a method to only those related to caches owned by a specific service by annotating the method parameter with the @ServiceName annotation.

The method below will only receive events for the caches owned by the service named `StorageService`:

```
@CoherenceEventListener
public void onEvent(@ServiceName("StorageService") EntryProcessorEvents event) {
    // TODO: process the event
}
```

Receive EntryProcessorEvents from a Specific Session

A typical use case is to obtain `NamedCache` and `NamedMap` instances from a `Session`. It is possible to restrict events received by a method to only those related to caches owned by a specific `Session` by annotating the method parameter with the @SessionName annotation.

The method below will only receive events for the caches owned by the `Session` named `BackEnd`:

```
@CoherenceEventListener
public void onEvent(@SessionName("BackEnd") EntryProcessorEvents event) {
    // TODO: process the event
}
```

3.6.2.8. Partition Level Transaction Events

A TransactionEvent is emitted in relation to all mutations in a single partition in response to executing a single request. These are commonly referred to as partition level transactions. For example, an `EntryProcessor` that mutates more than one entry (which could be in multiple caches) as part of a single invocation will cause a partition level transaction to occur encompassing all of those cache entries.

Transaction events are emitted by storage enabled cache services, they will only e received on the same member that the partition level transaction occurred.

To subscribe to `TransactionEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `TransactionEvent`.

For example, the `onEvent` method below will receive all transaction events emitted by storage

enabled cache services in the same JVM.

```
@CoherenceEventListener
public void onEvent(TransactionEvent event) {
    // TODO: process the event
}
```

Receive Specific TransactionEvent Types

There are a number of different `TransactionEvent` types.

- **Committing** - A COMMITTING event is raised prior to any updates to the underlying backing map. This event will contain all modified entries which may span multiple backing maps. Use the @Committing annotation
- **Committed** - A COMMITTED event is raised after any mutations have been committed to the underlying backing maps. This event will contain all modified entries which may span multiple backing maps. Use the @Committed annotation

To restrict the `TransactionEvent` types received by a method apply one or more of the annotations above to the method parameter. For example, the method below will receive `Committed` events.

```
@CoherenceEventListener
public void onEvent(@Committed TransactionEvent event) {
    // TODO: process the event
}
```

Receive TransactionEvent from a Specific Cache Service

Caches are owned by a Cache Service, it is possible to restrict events received by a method to only those related to caches owned by a specific service by annotating the method parameter with the @ServiceName annotation.

The method below will only receive events for the caches owned by the service named `StorageService`:

```
@CoherenceEventListener
public void onEvent(@ServiceName("StorageService") TransactionEvent event) {
    // TODO: process the event
}
```

3.6.2.9. Partition Transfer Events

A TransferEvent captures information concerning the transfer of a partition for a storage enabled member. Transfer events are raised against the set of BinaryEntry instances that are being transferred.

TransferEvents are dispatched to interceptors while holding a lock on the partition being transferred, blocking any operations for the partition. Event observer methods should therfore execute as quickly as possible of hand-off execution to another thread.

To subscribe to `TransferEvent` simply create a Spring bean with a listener method annotated with @CoherenceEventListener. The method should have a single parameter of type `TransferEvent`.

For example, the `onEvent` method below will receive all transaction events emitted by storage enabled cache services in the same JVM.

```
@CoherenceEventListener
public void onEvent(TransferEvent event) {
    // TODO: process the event
}
```

Receive Specific TransferEvent Types

There are a number of different `TransferEvent` types.

- **Arrived** - This `TransferEvent` is dispatched when a set of `BinaryEntry` instances have been transferred to the local member or restored from backup.The reason for the event (primary transfer from another member or restore from backup) can be derived as follows:

```
TransferEvent event;
boolean restored = event.getRemoteMember() == event.getLocalMember();
```

Use the @Arrived annotation to restrict the received events to arrived type.

- **Assigned** - This `TransferEvent` is dispatched when a partition has been assigned to the local member. This event will only be emitted by the ownership senior during the initial partition assignment. Use the @Assigned annotation to restrict received events.
- **Departing** - This `TransferEvent` is dispatched when a set of `BinaryEntry` are being transferred from the local member. This event is followed by either a `Departed` or `Rollback` event to indicate the success or failure of the transfer. Use the @Departing annotation to restrict received events.
- **Departed** - This `TransferEvent` is dispatched when a partition has been successfully transferred from the local member. To derive the `BinaryEntry` instances associated with the transfer, consumers should subscribe to the `Departing` event that would precede this event. Use the @Departed annotation to restrict received events.
- **Lost** - This `TransferEvent` is dispatched when a partition has been orphaned (data loss *may* have occurred), and the ownership is assumed by the local member. This event is only be emitted by the ownership senior. Use the @Lost annotation to restrict received events.
- **Recovered** - This `TransferEvent` is dispatched when a set of `BinaryEntry` instances have been recovered from a persistent storage by the local member. Use the @Recovered annotation to restrict received events.

- **Rollback** - This `TransferEvent` is dispatched when partition transfer has failed and was therefore rolled back. To derive the `BinaryEntry` instances associated with the failed transfer, consumers should subscribe to the `Departing` event that would precede this event. Use the `@Rollback` annotation to restrict received events.

To restrict the `TransferEvent` types received by a method apply one or more of the annotations above to the method parameter. For example, the method below will receive `Lost` events.

```
@CoherenceEventListener
public void onEvent(@Lost TransferEvent event) {
    // TODO: process the event
}
```

Multiple type annotations may be used to receive multiple types of `TransferEvent`.

Receive TransferEvent from a Specific Cache Service

Caches are owned by a Cache Service, it is possible to restrict events received by a method to only those related to caches owned by a specific service by annotating the method parameter with the `@ServiceName` annotation.

The method below will only receive events for the caches owned by the service named `StorageService`:

```
@CoherenceEventListener
public void onEvent(@ServiceName("StorageService") TransferEvent event) {
    // TODO: process the event
}
```

3.6.2.10. Unsolicited Commit Events

An `UnsolicitedCommitEvent` captures changes pertaining to all observed mutations performed against caches that were not directly caused (solicited) by the partitioned service. These events may be due to changes made internally by the backing map, such as eviction, or referrers of the backing map causing changes.

Unsolicited commit events are emitted by storage enabled cache services, they will only e received on the same member.

To subscribe to `UnsolicitedCommitEvent` simply create a Spring bean with a listener method annotated with `@CoherenceEventListener`. The method should have a single parameter of type `UnsolicitedCommitEvent`.

For example, the `onEvent` method below will receive all Unsolicited commit events emitted by storage enabled cache services in the same JVM.

```
@CoherenceEventListener
public void onEvent(UnsolicitedCommitEvent event) {
    // TODO: process the event
}
```

3.7. Filter Binding Annotations

Filter binding annotations are normal annotations that are themselves annotated with the @FilterBinding meta-annotation. A filter binding annotation represents a Coherence Filter and is used to specify a `Filter` in certain injection points, for example a View (CQC), `NamedTopic Subscriber` beans, event listeners, etc.

There are three parts to using a filter binding:

- The filter binding annotation
- An implementation of a FilterFactory that is annotated with the filter binding annotation. This is a factory that produces the required `Filter`.
- Injection points annotated with the filter binding annotation.

We will put all three parts together in an example. Let's use a Coherence `NamedMap` named `plants` that contains plants represented by instances of the `Plant` class as map values. Among the various properties on the `Plant` class there is a property called `plantType` and a property called `height`. In this exaple, we want to inject a view that only shows large palm trees (any palm tree larger than 20 meters). We would need a `Filter` that has a condition like the following: `plantType == PlantType.PALM && height >= 20`.

3.7.1. Create the filter binding annotation

First create a simple annotation, it could be called something like `PlantNameExtractor`

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

import com.oracle.coherence.spring.annotation.FilterBinding;

@FilterBinding                              ①
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface LargePalmTrees {          ②
}
```

① The annotation class is annotated with `@FilterBinding`

② The annotation name is `PlantNameExtractor`

In this case the annotation does not need any other attributes.

3.7.2. Create the `FilterFactory`

Now create the FilterFactory implementation that will produce instances of the required `Filter`.

```
import com.oracle.coherence.spring.annotation.FilterFactory;
import com.tangosol.util.Extractors;
import com.tangosol.util.Filter;
import com.tangosol.util.Filters;
import org.springframework.stereotype.Component;

@LargePalmTrees                                                          ①
@Component                                                               ②
public class LargePalmTreesFilterFactory<Plant>
        implements FilterFactory<LargePalmTrees, Plant> {
    @Override
    public Filter<Plant> create(LargePalmTrees annotation) {             ③
        Filter<Plant> palm = Filters.equal("plantType", PlantType.PALM);
        Filter<Plant> height = Filters.greaterEqual(
                Extractors.extract("height"), 20);
        return Filters.all(palm, height);
    }
}
```

① The class is annotated with the `PlantNameExtractor` filter binding annotation

② The class must be a Spring bean, let's annotate it with `@Component` so that component scanning will pick this class up as a Spring bean

③ The `create` method uses the Coherence `filters` API to create the required `filter`.

The parameter to the `create` method is the annotation used on the injection point. In this case the annotation has no values, but if it did we could access those values to customize how the filter is created.

For example, we can make the filter more general purpose by calling the annotation `@PalmTrees` and by adding a value parameter representing the height like this:

```
@FilterBinding
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface PalmTrees {
    String value();
}
@FilterBinding
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface PalmTrees {
    int value() default 0;
}
```

We then need to modify our filter factory to use the height value:

```
import com.oracle.coherence.spring.annotation.FilterFactory;
import com.tangosol.util.Extractors;
import com.tangosol.util.Filter;
import com.tangosol.util.Filters;
import org.springframework.stereotype.Component;

@PalmTrees                                                          ①
@Component                                                          ②
public class PalmTreesFilterFactory<Plant>
        implements FilterFactory<PalmTrees, Plant> {
    @Override
    public Filter<Plant> create(PalmTrees annotation) {             ③
        Filter<Plant> palm = Filters.equal("plantType", PlantType.PALM);
        Filter<Plant> height = Filters.greaterEqual(
                Extractors.extract("height"), annotation.value());    ④
        return Filters.all(palm, height);
    }
}
```

① The class is annotated with the more flexible `PalmTrees` filter binding annotation accepting a height parameter

② The class must be a Spring bean, let's annotate it with `@Component` so that component scanning will pick this class up as a Spring bean

③ The `create` method uses the Coherence `filters` API to create the required `filter`

④ Instead of hard-coding the height, we use the value from the `@PalmTrees` annotation

3.7.3. Annotate the Injection Point

Now the application code where the view is to be injected can use the custom filter binding annotation.

```
    @View                                                           ①
    @PalmTrees(1)                                                   ②
    @Name("plants")                                                 ③
    @Resource(name = "getCache")                                    ④
    private NamedMap<Long, Plant> palmTrees;
```

① The `@View` annotation indicates that this is a view rather than a plain `NamedMap`

② The `@PalmTrees` annotation links to the custom filter factory which is used to create the filter for the view. The annotation value of `1` indicates that we are interested in all palm trees of at least 1 meter in height.

③ The `@Name` annotation indicates the underlying cache/map name to use for the view

④ Due to some Spring limitations, we have to use the `@Resource` annotation to inject the `NamedMap` as we need to match the underlying bean by name.

3.8. Extractor Binding Annotations

ValueExtractor binding annotations are normal annotations that are themselves annotated with the @ExtractorBinding meta-annotation. An extractor binding annotation represents a Coherence ValueExtractor and is used to specify a `ValueExtractor` in certain injection points, for example a View (CQC), `NamedTopic Subscriber` beans, `MapEvent` listeners, etc.

There are three parts to using an extractor binding:

- The extractor binding annotation
- An implementation of a ExtractorFactory that is annotated with the extractor binding annotation. This is a factory that produces the required `ValueExtractor`.
- Injection points annotated with the extractor binding annotation.

As an example, let's continue with our previous example, where we have a Coherence `NamedMap` named `plants` that contains `Plant` instances as values. In this example we are interested in inject a map of plant names instead of the actual plant instances. Each plant has a `name` property that we will use for that purpose. We will need a `ValueExtractor` that extracts the `name` property and the resulting map of plant names can be injected into our Spring beans.

3.8.1. Create the extractor binding annotation

First create a simple annotation called `PlantName`

```
@ExtractorBinding                          ①
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface PersonAge {              ②
}

import com.oracle.coherence.spring.annotation.ExtractorBinding;
import com.oracle.coherence.spring.annotation.FilterBinding;

import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@ExtractorBinding                                ①
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface PlantNameExtractor {           ②
}
```

① The annotation class is annotated with `@ExtractorBinding`

② The annotation name is `PlantNameExtractor`

In this case the annotation does not need any other attributes.

3.8.2. Create the `ExtractorFactory`

Now create the ExtractorFactory implementation that will produce instances of the required `ValueExtractor`.

```
import com.oracle.coherence.spring.annotation.ExtractorFactory;
import com.tangosol.util.Extractors;
import com.tangosol.util.ValueExtractor;
import org.springframework.stereotype.Component;

@PlantNameExtractor                                                    ①
@Component                                                             ②
public class PlantNameExtractorFactory<Plant>
        implements ExtractorFactory<PlantNameExtractor, Plant, String> {
    @Override
    public ValueExtractor<Plant, String> create(PlantNameExtractor annotation) {  ③
        return Extractors.extract("name");
    }
}
```

① The class is annotated with the `PlantNameExtractor` extractor binding annotation

② The class must be a Spring bean, let's annotate it with `@Component` so that component scanning will pick this class up as a Spring bean

③ The `create` method uses the Coherence `Extractors` API to create the required extractor, in this case a trivial property extractor.

The parameter to the `create` method is the annotation used on the injection point. In this case the annotation has no values, but if it did we could access those values to customize how the `ValueExtractor` is created.

3.8.3. Annotate the Injection Point

Now the application code where the view is to be injected can use the custom extractor binding annotation.

```
@View               ①
@PersonAge          ②
@Name("people")     ③
private NamedMap<String, Integer> ages;   ④
    @View                                                              ①
    @PlantNameExtractor                                                ②
    @Name("plants")                                                    ③
    @Resource(name = "getCache")                                       ④
    private NamedMap<Long, String> plants;                             ⑤
```

① The `@View` annotation indicates that this is a view rather than a plain `NamedMap`

② The `@PlantNameExtractor` annotation links to the custom extractor factory used to create the

`ValueExtractor` for the view

③ The `@Name` annotation indicates the underlying cache/map name to use for the view

④ Due to some Spring limitations, we have to use the `@Resource` annotation to inject the `NamedMap` as we need to match the underlying bean by name.

⑤ Note that the `NamedMap` generics are now `Long` and `String` instead of `Long` and `Plant` as the `Plant` values from the underlying cache are transformed into `String` values by extracting just the name property.

Chapter 4. Coherence Spring Cache

This section dives into the Coherence Spring Cache module. It explains how to use Coherence's support for the Spring Framework's Cache Abstraction.

4.1. Introduction

Spring provides its own cache abstraction, allowing you to add caching to Java methods. Coherence Spring provides an implementation of this abstraction for Oracle Coherence.

Spring's Cache abstraction also supports JSR-107 which is also supported by Oracle Coherence. As such you have another alternative for setting up caching.

If you are using JPA/Hibernate you may also consider using the Coherence support for Hibernate's second-level cache SPI, which is provided by the Coherence Hibernate project.

4.2. Configuring Coherence Cache for Spring

As a start, please familiarize yourself with Spring's Cache Abstraction by reading the relevant section of Spring's reference documentation.

Properties

```
```

Yaml

```
example:
  property:
    alpha: a
```

Properties

```
```

Yaml

```
spring:
  devtools:
    restart:
      exclude: "static/**,public/**"
```

Example 4. Creating a CoherenceInstance

Java

```
@Configuration
@EnableCaching
public class CacheConfiguration {

    @Bean
    public CoherenceInstance coherenceInstance() {
      return new CoherenceInstance();
    }

    @Bean
    public CacheManager cacheManager(CoherenceInstance coherenceInstance) {
      return new CoherenceCacheManager(coherenceInstance);
    }
}
```

XML

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

  <cache:annotation-driven/>

  <bean id="coherenceInstance"
class="com.oracle.coherence.spring.CoherenceInstance"/>

  <bean id="cacheManager"
class="com.oracle.coherence.spring.cache.CoherenceCacheManager">
    <constructor-arg ref="coherenceInstance"/>
  </bean>
</beans>
```

Chapter 5. Coherence Spring Session

This section dives into the Coherence Spring Session module. It explains how to use Coherence's support for Spring Session.

5.1. Getting Started

TBD

Chapter 6. Coherence Spring Data

This section dives into the Coherence Spring Data module. It explains how to use Coherence's support for Spring Data repositories.

6.1. Getting Started

TBD

Chapter 7. Coherence Spring Boot

This section dives into the Coherence Spring Boot module. It explains how to use Coherence's dedicated support for Spring Boot, e.g. Autoconfiguration.

7.1. Getting Started

In order to start using Coherence with Spring Boot you have to add the `coherence-spring-boot-starter` dependency as well as the desired version of Coherence.

Maven

```
<dependencies>
    <dependency>
        <groupId>com.oracle.coherence.spring</groupId>
        <artifactId>coherence-spring-boot-starter</artifactId>
        <version>3.0.0-SNAPSHOT</version>
    </dependency>
    <dependency>
        <groupId>com.oracle.coherence.ce</groupId>
        <artifactId>coherence</artifactId>
        <version>21.06-M1</version>
    </dependency>
</dependencies>
```

Gradle

```
dependencies {
    compile("com.oracle.coherence.spring:coherence-spring-boot-starter:3.0.0-
SNAPSHOT")
    compile("com.oracle.coherence.ce:coherence:21.06-M1")
}
```

As Coherence Spring takes advantage of the new Coherence Bootstrap API, it requires Oracle Coherence CE version `20.12` or higher.

7.2. Using Coherence with Spring Boot

By adding the `coherence-spring-boot-starter` dependency, AutoConfiguration will be activated via the `CoherenceAutoConfiguration` class. This will also bind the `CoherenceProperties` for further configuration. The configuration for Spring Boot's Coherence support may look like the following:

Example YAML configuration (Properties)

```
```

Example YAML configuration (Yaml)

```
coherence:
  logging:
    destination: slf4j
    logger-name: MyCoherence
  sessions:
  - name: default
    config: "coherence-cache-config.xml"
    priority: 1
  - name: test
    config: "test-coherence-config.xml"
    priority: 2
  properties:
    coherence.log.limit: 400
    coherence.log.level: 1
```

The following configuration properties are available.

Table 1. Coherence Configuration Properties

Key	Default Value	Description
coherence.logging.destination		The type of the logging destination. Default to `slf4j` if not set.
coherence.logging.severity-level		Specifies which logged messages are emitted to the log destination. The legal values are `-1` to `9`. No messages are emitted if `-1` is specified. More log messages are emitted as the log level is increased.
coherence.logging.logger-name		
coherence.logging.message-format		
coherence.logging.character-limit		
coherence.properties.*		Any native Coherence properties
coherence.sessions[0].name		
coherence.sessions[0].type		Represents the various session type that can be configured: CLIENT, SERVER, GRPC

Key	Default Value	Description
coherence.sessions[0].config		The Coherence cache configuration URI for the session
coherence.sessions[0].priority		The priority order to be used when starting the session. Sessions will be started with the lowest priority first.
coherence.sessions[0].scope-name		The scope name for the session.

All but the session property are translated into native Coherence properties. If both Spring Boot property AND a native property `coherence.properties.*` are configured, the Spring Boot property is used.

For a list of available native properties, please consult the reference guide chapter on System Property Overrides.

7.3. Customize Coherence

7.4. Coherence Support of the Spring Boot ConfigData API

Starting with Spring Boot `2.4.x` you can define your own custom config locations. This allows you to import these as property sources. As such, Coherence Spring allows you to use a Coherence cluster as a source of configuration data for your Spring Boot based applications.

Please also consult the Spring Boot reference guide on Externalized Configuration, especially the chapter on Importing Additional Data.

Please also see the chapter on Coherence Spring Cloud Config.

TBD

7.5. Using Coherence as Spring Caching Provider

If caching is enabled via `@EnableCaching`, Coherence Autoconfiguration will it automatically provide a `CacheManager` to the ApplicationContext, however only if no `CacheManager` was configured explicitly beforehand.

Chapter 8. Coherence Spring Cloud Config

This section explains how to configure Coherence using Spring Cloud Config. Furthermore, this chapter also shows how to use Coherence as a Spring Cloud Config storage backend, allowing you to set up Spring applications with configuration data stored in Coherence.

8.1. Overview

Spring Cloud Config provides support for externalized configuration in distributed systems. It integrates seamlessly with Spring Boot applications and allows you to externalize / centralize critical application properties. Spring Cloud Config provides numerous storage backends for your configuration data and as part of Coherence Spring we also provide a backend for Oracle Coherence.

Please familiarize yourself with the Spring Cloud Config reference documentation.

In this chapter we will cover two aspects of Coherence-specific support for Spring Cloud Config:

- Configure Coherence and its Spring support using Spring Cloud Config
- Use Oracle Coherence as a configuration backend for Spring Cloud Config and thus store your Configuration data in a Coherence cluster

Let's get started with an example to show the general functioning of Spring Cloud Config.

8.2. Demo

This demo is essentially the same as is used in the Quickstart chapter. However, we externalize some Coherence configuration using Spring Cloud Config. The source code for the demo is part of the Coherence Spring source code repository. Therefore, to get started, please clone its repository:

Clone the Spring Cloud Config demo project

```
$ git clone https://github.com/coherence-community/coherence-spring.git
$ cd coherence-spring
```

You now have checked out all the code for Coherence Spring. The relevant demo code for the Spring Cloud Config demo is under `coherence-spring-samples/coherence-spring-cloud-config-demo/`. The demo consists of 2 Maven modules:

- **coherence-spring-cloud-config-demo-server**: Spring Cloud Config Server implementation
- **coherence-spring-cloud-config-demo-app**: Main application

The Config Server is essentially using 2 dependencies:

Maven

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>    ①
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>  ②
</dependency>
```

① Spring Cloud Config Server dependency

② Provides rudimentary security for the exposed configuration REST endpoints using Spring Security

The demo client on the other hand will use the following dependencies:

Maven

```
<dependency>
    <groupId>com.oracle.coherence.spring</groupId>
    <artifactId>coherence-spring-boot-starter</artifactId> ①
    <version>{coherence-spring.version}</version>
</dependency>
<dependency>
    <groupId>com.oracle.coherence.ce</groupId>
    <artifactId>coherence</artifactId>                     ②
    <version>{coherence.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>   ③
</dependency>
```

① Provides all integration code, caching + autoconfiguration support

② The Oracle Coherence dependency

③ The dependency to integrate with the Spring Cloud Config server

We made the decision to not automatically bring in the Coherence dependencies. The main reason is that users can specify the version they need, either the Oracle Coherence CE (OSS) or the commercial version.

8.2.1. Configure the Demo Application

In order to run the demo, we first need to create a Git repository that will contain the configuration data.

Setup the Config Data

```
$ cd /path/to/git/repo
$ mkdir coherence-spring-config-repository
$ cd coherence-spring-config-repository
$ git init
```

Add a properties file called `config-client.properties`:

config-client.properties

```
coherence.logging.severity-level=6
①
coherence.logging.destination=slf4j
②

coherence.properties.coherence.cluster=Demo Cluster
③
coherence.properties.coherence.member=Demo Cluster Member
④
coherence.properties.coherence.management.remote=true
⑤
coherence.properties.coherence.management=all
⑥
coherence.properties.coherence.management.report.autostart=true
⑦
coherence.properties.coherence.reporter.output.directory=/path/to/reports/
⑧
coherence.properties.coherence.management.report.configuration=/reports/report-
all.xml ⑨
```

① `-1` emits no log messages, `9` emits the most

② Specifies the logger e.g. `stdout`, `log4j`, `log4j2`, `slf4j`

③ The name of the cluster

④ The name of the cluster member

⑤ Specifies whether this cluster node exposes its managed objects to remote MBean server. `true` or `false`

⑥ `none` means no MBean server is instantiated. `all` enables management of both local and remotely manageable cluster nodes.

⑦ `true` or `false` (default) Specifies whether the Reporter automatically starts when the node starts.

⑧ The output directory for generated reports. By default, reports are saved reports to the directory from which the cluster member starts.

⑨ You can control which reports are generated by specifying a different report group configuration file. The pre-defined reports are located at `coherence-21.06-M1.jar/reports`

For more options please see the following three chapters in the official Oracle Coherence reference

guide:

- Operational Configuration Elements
- System Property Overrides
- Using Oracle Coherence Reporting

8.2.2. Run the Demo Application

Please execute the following:

Start the Spring Cloud Config Server

```
 $ ./mvnw clean package -pl :coherence-spring-cloud-config-demo-server
 $ cd coherence-spring-samples/coherence-spring-cloud-config-demo/coherence-spring-
cloud-config-demo-server/target
 $ java -jar coherence-spring-cloud-config-demo-server-3.0.0-SNAPSHOT.jar \n
     --spring.cloud.config.server.git.uri=file:///path/to/git/repo
```

Start the Coherence Spring Application

```
 $ ./mvnw clean package -pl :coherence-spring-cloud-config-demo-app
 $ cd coherence-spring-samples/coherence-spring-cloud-config-demo/coherence-spring-
cloud-config-demo-app/target
 $ java -jar coherence-spring-cloud-config-demo-app-3.0.0-SNAPSHOT.jar
```

Feel free to change configuration settings and see, once you restart the apps, how the behavior of the Coherence cluster changes.

8.3. Use Spring Cloud Config Server to Configure Coherence

The previously discussed demo application illustrated the main concepts of using Spring Cloud Config Server as a configuration backend for Oracle Coherence. For a general understanding of Spring Cloud Config Server, please consult the respective reference documentation.

Coherence Spring is essentially unaware of Spring Cloud Config Server. Coherence Spring merely takes advantage of Spring Boot's configuration facilities. The main integration point for configuration between Spring and Oracle Coherence is the `SpringSystemPropertyResolver` class, which makes the properties of Spring's `Environment` available to Oracle Coherence.

When using Spring Boot (and not just plain Spring Framework), we also provide the `CoherenceProperties` class. It provides means to expose Coherence Spring configuration options in a type-safe manner, to provide code completion via your IDE etc.

Providing dedicated CoherenceProperties support is work in progress.

Behind the scenes using `CoherenceProperties.getCoherencePropertiesAsMap()` will translate the

explicit Spring Boot properties into the property format used by Oracle Coherence. It is important to note that you can always provide ANY Oracle Coherence property via the `coherence.properties.*` prefix.

For instance the following properties are equivalent:

Equivalent Properties

```
coherence.logging.severity-level=5
coherence.logging.destination=log4j

coherence.properties.coherence.log.level=5
coherence.properties.coherence.log=log4j
```

Please also see Coherence Support of the Spring Boot ConfigData API.

8.4. Coherence as Spring Cloud Config Server Backend

TBD

Chapter 9. Appendices