Coherence Spring Reference Documentation

Gunnar Hillert, Ryan Lubke, Vaso Putica

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.

Another great example application is the Spring Boot implementation of the To-do List application.

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 `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 samples/coherence-spring-demo/coherence-spring-demo-boot
```

Now we are ready to run the application:

Run the Spring Boot application

```
$ java -jar 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 inspecting the cache using the open-source tool VisualVM.

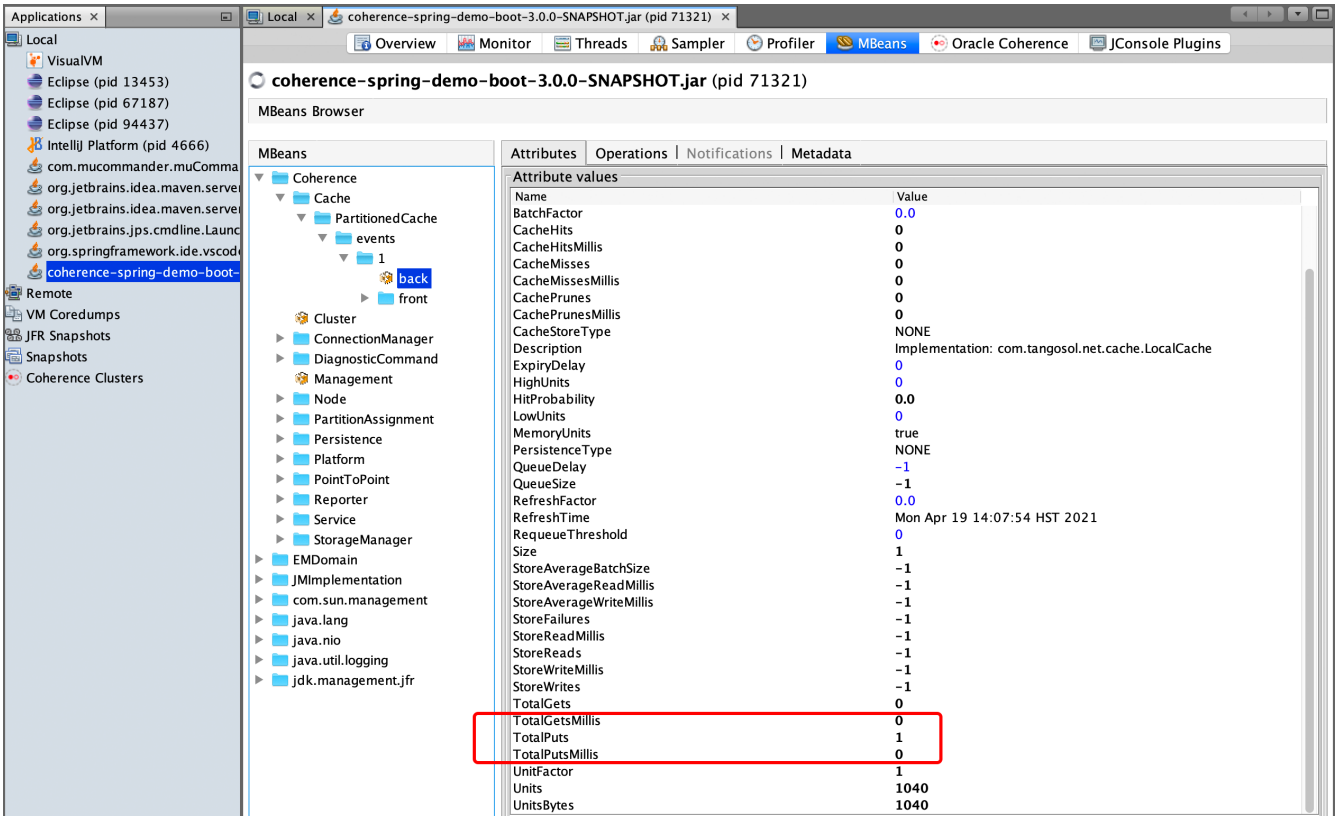

Figure 1. VisualVM - Cache Put

Under the `MBeans` tab you will find the Cache MBeans, including and entry for the `events` cache, providing numerous statistical information regarding the cache.

Retrieving Cache Statistics

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

You should see an entry for `TotalPuts` of `1`.

When using VisualVM consider installing the respective Coherence VisualVM Plugin as it provides some additional insights and visualizations.

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, this time via the Coherence VisualVM Plugin:

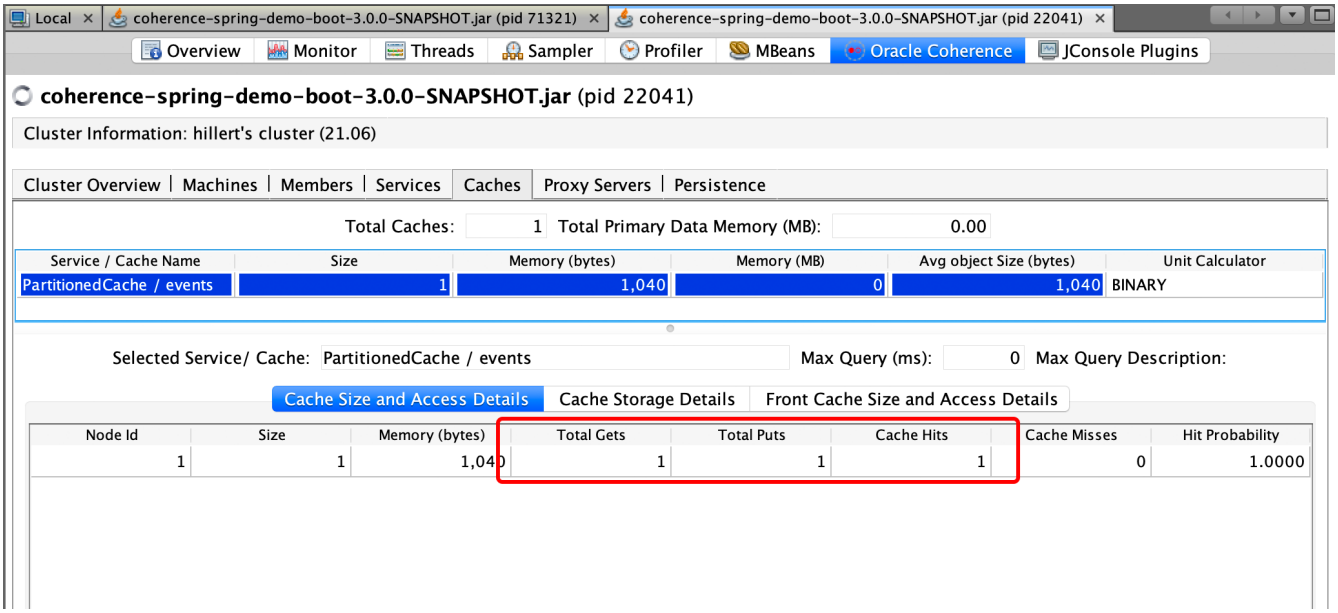

Figure 2. Cache Statistics via Coherence VisualVM Plugin

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</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</version>
</dependency>
```

Gradle

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

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

Coherence Spring uses 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` class under the covers. Therefore, you can alternatively also declare `@Import(CoherenceSpringConfiguration.class)` instead.

In most use-cases, only a single Coherence Session is expected to be used. Therefore, without providing any further configuration the default session is configured using the embedded default configuration file. This results in the application joining Coherence as a cluster member (Session type `SERVER`). This is of course not the only way. Coherence Spring support the following 3 session types:

- **SERVER** - Join as Coherence cluster member. This is the default session type.
- **CLIENT** - Connect to Coherence as a Coherence*Extend client
- **GRPC** - Connect to Coherence as gRPC client

If the application is a Coherence cluster member, or a Coherence*Extend client, then all that needs to be specified is the Coherence configuration file name. For instance, you may for example provide an implementation of the `AbstractSessionConfigurationBean`, to specify the type of your session and

to use a custom Coherence configuration file.

SessionConfigurationBean

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

If you connect as gRPC client, however, the properties change slightly and you need to specify a `GrpcSessionConfigurationBean`:

GrpcSessionConfigurationBean

```
@Bean
GrpcSessionConfigurationBean grpcSessionConfigurationBean() {
    final GrpcSessionConfigurationBean sessionConfigurationBean = new
GrpcSessionConfigurationBean();
    sessionConfigurationBean.setName("sessionName");
    sessionConfigurationBean.setChannelName("grpcChanelBeanName");
    return sessionConfigurationBean;
}
```

The Channel Name property would refer to a grpcChannel bean.

3.4. Configure Multiple Sessions

If you need to configure multiple Coherence sessions, simply define multiple `SessionConfigurationBeans`. The auto-configuration will pick those up automatically to configure the required sessions.

The `default` session will only exist when zero sessions are specifically configured, or the default session is specifically configured with the default session name.

3.5. Session Configuration Bean Properties

Depending on the session type the available properties change a bit. The following properties all to ALL session types.

name

The name of the session. If not set, it will be set to the default session name which is an empty String.

scopeName

A scope name is typically used in an application where the Coherence cluster member has multiple sessions. The scope name is used to keep the sessions separate. The scope name will be applied to the session's underlying *ConfigurableCacheFactory* and used to scope Coherence services. In this way multiple session configurations may use identical service names, which will be kept separate using the scope. On a Coherence cluster member, each session should have a unique scope name.

type

The session type of this configuration. There are three different types of sessions that can be configured:

- **server** represents storage enabled cluster member session.
- **client** represents a storage disabled cluster member or Coherence*Extend client session.
- **grpc** is a gRPC client session (see the gRPC documentation).

The type of the session affects how the bootstrap API starts the session.

priority

The priority specifies the order to use, when starting the session. Sessions will be started with the lowest priority first. If this property is not specified, the property will default to `0`.

The following property applies to the `CLIENT` (Coherence*Extend) and `Server` mode, only:

configUri

The Coherence cache configuration URI for the session. As already mentioned, the most common configuration to set will be the Coherence configuration file name. If not specified, the default value will be `coherence-cache-config.xml`.

The following property applies to the `GRPC` mode, only:

channelName

Sets the underlying gRPC channel. If not set, it will default to `localhost` and port `1408`.

serializer

Specifies the serializer to that shall be used, in order to serialize gRPC message payloads. If not specified, the serializer will be the default Coherence serializer, either POF if it has been enabled with the `coherence.pof.enabled` system property or Java serialization.

tracingEnabled

Specifies if client gRPC tracing should be enabled. This is `false` by default.

3.6. Dependency Injection

Coherence Spring provides comprehensive support for the injection Coherence objects into your Spring beans including: `Session`, `NamedMap`, `NamedCache`, `ContinuousQueryCache`, `ConfigurableCacheFactory`, `Cluster`.

For the most part, you can use the equivalent Coherence Spring annotation that match the annotations from Coherence's CDI or Micronaut support.

3.6.1. Injecting NamedMap and NamedCache

Coherence `NamedMap` and `NamedCache` instances can be injected as beans in Spring applications. The mechanics of injecting `NamedMap` or `NamedCache` beans is identical, so any use of `NamedCache` in the examples below can be replaced with `NamedMap`. Other more specialized forms of `NamedMap` and `NamedCache` can also be injected, for example the asynchronous forms of both classes and views.

In Spring one caveat exists regarding the injection of Map-based classes that directly inherit from `java.util.Map` including `NamedCache` and `NamedMap` when using the `@Autowired` annotation. Instead of injecting actual instances of Beans representing a `java.util.Collection` or `java.util.Map`, Spring will inject a collection of all the beans that represent the specified bean type instead. As a work-around, you can use the `@Resource` annotation, but it has its own limitations, for instance, not being usable for constructor injection.

Example of using the @Resource annotation

```
@Resource(name = COHERENCE_CACHE_BEAN_NAME)
private NamedCache numbers;                    ①

@Resource(name = COHERENCE_CACHE_BEAN_NAME)
@Name("numbers")                               ②
private NamedCache namedCache;
```

① If not specified, the name of the field will be used to determine the cache name

② Alternatively, you can specify the name of the cache using the `@Name` annotation

For more information, please see Fine-tuning Annotation-based Autowiring with Qualifiers in the Spring Framework reference guide.

In order to provide a better user-experience around the dependency injection of maps and caches, Coherence Spring introduces its own set of annotations. The following annotations are available:

- @CoherenceCache
- @CoherenceMap
- @CoherenceAsyncCache
- @CoherenceAsyncMap

Using these annotations, you can inject any Coherence NamedMap and NamedCache in any situation including constructors.

Furthermore, the annotations also give you some added conveniences such as the ability to specify the name of the cache, or the name of the Coherence session as part of the annotation. E.g., the above example can be simplified to:

Example of using the @CoherenceCache annotation

```
    @CoherenceCache
    private NamedCache numbers;                 ①

    @CoherenceCache("numbers")                  ②
    private NamedCache namedCache;
```

① If not specified, the name of the field will be used to determine the cache name

② Alternatively, you can specify the name of the cache using the `@Name` annotation

3.6.1.1. Specify the Map/Cache Name

As already mentioned above, you specify the name of the map/cache using the value-property of the annotation. Of course, the same applies when injecting a constructor or method parameter:

Example of constructor injection of a NamedMap

```
@Service
public class SomeService {

    public SomeService(@CoherenceMap("people") NamedMap<String, Person> map) {
        // TODO: initialize the service...
    }
}
```

If injecting a cache/map via the constructor, AND you do not specify a cache/map name, then Coherence Spring will try to derive the name of the cache/map from the parameter name. However, this only works if either the compiler flag `-parameters` (Java 8+) is enabled, or if the JVM generates debugging info. For more information see the article Method Parameter Reflection in Java.

If you prefer, you can also specify the name of the map/cache using the `@Name` annotation. The example below will inject a `NamedMap` that uses an underlying cache named people:

Example of using the @Name annotation

```
@CoherenceMap
@Name("people")
private NamedMap<String, Person> map;
```

3.6.1.2. Specify the Owning Session Name

Whilst most applications probably use a single Coherence Session, there are uses-cases where an

application may have multiple sessions. In this case, when injecting for example a `NamedMap`, the specific session can be specified by annotating the injection point with either `@SessionName` or more concise with the `session` parameter available for the following annotations:

- @CoherenceCache
- @CoherenceMap
- @CoherenceAsyncCache
- @CoherenceAsyncMap

In the previous examples where no separate `Session` name was specified, Coherence will use the default session to obtain the caches/maps. Assuming that the application has multiple sessions configured, one of which is named *Catalog*, the following example injects a `NamedMap` from an underlying cache named *products* in the *Catalog* session.

Example of using the @SessionName annotation

```
@CoherenceMap
@SessionName("Catalog")
@Name("products")
private NamedMap<String, Product> map;
```

This can be further streamlined to:

Example of using the @CoherenceMap annotation with the session parameter

```
@CoherenceMap(name="products", session="Catalog")
private NamedMap<String, Product> map;
```

The same annotation can be used on method parameter injection points as well:

Example of using @CoherenceMap with session parameter in a constructor

```
@Controller
public class CatalogController {

    public CatalogController(@CoherenceMap(name="products", session="Catalog")
                             NamedMap<String, Product> products) {
        // TODO: initialize the bean...
    }
}
```

3.6.2. Injecting AsyncNamedMap & AsyncNamedCache

It is possible to inject the asynchronous classes `AsyncNamedMap` and `AsyncNamedCache` as beans in exactly the same way as described above. Just change the type of the injection point to be `AsyncNamedMap` or `AsyncNamedCache` using one of the following annotations:

- @CoherenceAsyncCache

- @CoherenceAsyncMap

Injecting an AsyncNamedMap

```
@CoherenceAsyncMap("people")
private AsyncNamedMap<String, Person> map;
```

3.6.3. Injecting Views (CQC)

View (or ContinuousQueryCache) beans can be injected by specifying the @View annotation at the injection point. A view is a sub-set of the data in an underlying cache, controlled by a Filter.

Injecting an AsyncNamedMap

```
@CoherenceMap("people")
@View                                                ①
private NamedMap<String, Person> map;
```

① The injection point has been annotated with `@View`, so the injected `NamedMap` will actually be an implementation of a `ContinuousQueryCache`.

In the above example, no `Filter` has been specified, so the default behaviour is to use an AlwaysFilter. This means that the view will contain all the entries from the underlying cache (typically a distributed cache). As a `ContinuousQueryCache` will hold keys and values locally in deserialized form, this can often be a better approach than using a replicated cache.

3.6.3.1. Specify a View Filter

Filters are specified for views using a special filter binding annotation. These are annotations that are themselves annotated with the meta-annotation @FilterBinding. Coherence Spring comes with some built in implementations, for example @AlwaysFilter and @WhereFilter. It is simple to implement custom Filters as required by applications (see the Filter Binding Annotation section for more details).

For example, if there was a cache named "people", containing `Person` instances, and the application required a view of that cache to just contain `People` where the "lastName" attribute is equal to "Simpson", then the `@WhereFilter` filter binding annotation could be used to specify the `Filter`. The `@WhereFilter` annotation produces a `Filter` created from a Coherence CohQL where-clause, in this case `lastName == 'Simpson'`.

Injecting a @CoherenceMap with @WhereFilter

```
    @CoherenceMap("people")                          ①
    @View                                            ②
    @WhereFilter("lastName = 'Simpson'")             ③
    private NamedMap<String, Person> allSimpsons;    ④
```

① The name of the underlying map for the view is "people".

② The `@View` annotation specifies that a view will be injected rather than a raw `NamedMap`.

③ The `@WhereFilter` annotation specifies the CohQL expression.

④ The `NamedMap` contains only people with the last name `Simpson`.

The above CohQL expression is still rather simple. Let's further restrict the results:

@WhereFilter with a more complex CohQL expression

```
@CoherenceMap("people")
@View
@WhereFilter("lastName = 'Simpson' and age > 10") ①
private NamedMap<String, Person> simpsons;
```

① The @WhereFilter also filters on the `age` property.

The view injected above will be all People with a `lastName` attribute equal to `Simpson` and an `age` attribute greater than `10`.

The Coherence reference guide has an in-depth chapter on CohQL and more details on the *WHERE* clause under Filtering Entries in a Result Set

Other built-in or custom filter binding annotations can be combined as well and multiple filter-binding annotations can be added to the same injection point to build up more complex views. The Filter instances produced from each filter binding annotation will all be collected together in an AllFilter, which will logically combine them together.

3.6.3.2. Specify a View Transformer

The values in a view map do not have to be the same as the values in the underlying cache. Instead, a ValueExtractor can be used to transform the actual cache value into a different value in the view. ValueExtractors are specified for views using a special extractor binding annotation. These are annotations that are themselves annotated with the meta-annotation @ExtractorBinding. The Coherence Spring framework comes with some built in implementations, for example @PropertyExtractor, and it is simple to implement other as required by applications (see the Extractor Binding Annotation section for more details).

For example, if there was a cache named "people", containing Person instances, and the application required a view where the value was just the age attribute of each Person rather than the whole cache value. A @PropertyExtractor annotation could be used to specify that the values should be transformed using a property extractor.

Injecting a @CoherenceMap with multiple @WhereFilter

```
@CoherenceMap("people")                    ①
@View                                      ②
@PropertyExtractor("age")                  ③
private NamedMap<String, Integer> ages;    ④
```

① The name of the underlying map for the view is "people".

② The `@View` annotation specifies that a view will be injected rather than a raw `NamedMap`.

③ The `@PropertyExtractor` annotation specifies that a `ValueExtractor` should be used to transform the underlying cache values into different values in the view. In this case the `@PropertyExtractor` annotation will produce a value extractor to extract the `age` property.

④ Note that the map injected is now a `NamedMap<String, Integer>` with generic types of `String` and `Integer` because the values have been transformed from `Person` to `Integer`.

Multiple extractor bindings can be applied to the injection point, in which case the view value will be a `List` of the extracted attributes.

3.6.4. Injecting a Session

Sometimes it might not be possible to inject a Coherence resource, such as `NamedMap` or `NamedCache` directly because the name of the resource to be injected is not known until runtime. In this case it makes sense to inject a `Session` instance which can then be used to obtain other resources.

The simplest way to inject a `Session` is to annotate a field, method parameter, or other injection point with your preferred Spring-supported injection annotation such as `@Autowired` or `@Inject`:

Injecting a Coherence Session instance

```
@RestController
public class MyBean {
    @Inject                                     ①
    private Session session;
```

① Other injection annotations such as `@Autowired` can be used as well

Injecting a Coherence Session using constructor injection

```
@RestController
public class MyBean {
    @Autowired                                  ①
    public MyBean(Session session) {
        // TODO...
    }
}
```

① If your class has only a single constructor, you can even omit the `@Autowired` annotation

Both examples above will inject the default `Session` instance into the injection point.

3.6.4.1. Specify a Session Name

For most applications that only use a single `Session` the simple examples above will be all that is required. Some applications though may use multiple named Session instances, in which case the `Session` name need to be specified. This can be done by adding the `@Name` annotation to the injection point.

Injecting a specific (named) Coherence Session

```
@RestController
public class MyBean {
    @Autowired                                  ①
    @Name("Catalog")
    private Session session;
}
```

① Other injection annotations such as `@Inject` can be used as well

or into a constructor:

Injecting a specific (named) Coherence Session via constructor

```
@RestController
public class MyBean {
    @Autowired                                  ①
    public MyBean(@Name("Catalog") Session session) {
        // TODO...
    }
}
```

① If your class has only a single constructor, you can even omit the `@Autowired` annotation

3.6.5. Injecting NamedTopic

Coherence NamedTopic instances can be injected as beans in Spring applications.

An alternative way to write message driven applications instead of directly injecting `NamedTopic`, `Publisher` or `Subscriber` beans is to use Messaging with Coherence Topics.

3.6.5.1. Injecting NamedTopic

The simplest way to inject a `NamedTopic` is to just annotate the injection point with `@javax.inject.Inject`.

Inject NamedTopic

```
@Inject
private NamedTopic<Person> people;
```

In this example the injection point field name is used to determine the topic name to inject, so a `NamedTopic` bean with an underlying topic name of `people` will be injected.

As an alternative to using a `NamedTopic` directly in code, Coherence Spring also supports annotating methods directly as publishers and subscribers. See the Messaging with Coherence Topics section of the documentation.

Specify the Topic Name

Sometimes the name of the topic being injected needs to be different to the injection point name. This is always the case when injecting into method parameters as the parameter names are lost by the time the injection point is processed. In this case we can use the @Name annotation to specify the underlying cache name.

The example below will inject a `NamedTopic` that uses an underlying topic named `orders`.

Use @Name to specify topic name

```
@Inject
@Name("people")
private NamedTopic<Order> orders;
```

The same applies when injecting a constructor or method parameter:

Use @Name to specify topic name on a parameter

```
@Singleton
public class SomeBean {
    @Inject
    public SomeBean(@Name("orders") NamedTopic<Order> topic) {
    // ToDo:
    }
}
```

Specify the Session Name

Whilst most applications probably use a single Coherence Session there are uses-cases where an application may have multiple sessions. In this case, when injecting a `NamedTopic` the specific session can be specified by annotating the injection point with @SessionName.

In the previous examples where no `@SessionName` was specified Coherence will use the default session to obtain the caches.

For example, assume the application has multiple sessions configured, one of which is named `Customers`. The following code snippet injects a `NamedTopic` using an underlying topic named `orders` in the `Customers` session.

Use @SessionName to specify session

```
@Inject
@SessionName("Customers")
@Name("orders")
private NamedTopic<Order> topic;
```

Again, the same annotation can be used on method parameter injection points.

Use @SessionName to specify session on a method parameter

```
@Controller
public class OrderProcessor {
    @Inject
    public OrderProcessor(@SessionName("Customers") @Name("orders")
                          NamedTopic<Order> orders) {
        // ToDo:
    }
}
```

3.6.5.2. Injecting a NamedTopic Publisher

If application code only needs to publish messages to a Coherence `NamedTopic` then instead of injecting a `NamedTopic` bean, a Publisher bean can be injected.

The simplest way to inject a `Publisher` is just to annotate the injection point of type `Publisher` with `@Inject`, for example:

Inject a Publisher

```
@Inject
private Publisher<Order> orders;
```

The example above will inject a `Publisher` bean, the name of the underlying `NamedTopic` will be taken from the name of the injection point, in this case `orders`.

Specify the Topic Name

If the name of the injection point cannot be used as the `NamedTopic` name, which is always the case with injection points that are method or constructor parameters, then the @Name annotation can be used to specify the topic name.

For example, both of the code snippets below inject a `Publisher` that published to the `orders` topic:

Inject a Publisher that publishes on the `orders` topic

```
@Inject
@Name("orders")
private Publisher<Order> orders;
```

Inject a Publisher that publishes on the `orders` topic

```
@Controller
public class OrderController {
    @Inject
    public OrderController(@Name("orders") Publisher<Order> topic) {
        // ToDo:
    }
}
```

Specify the Owning Session

As with injection of `NamedTopics`, in applications using multiple `Session` instances, the name of the `Session` that owns the underlying `NamedTopic` can be specified when injecting a `Publisher` by adding the `@SessionName` annotation.

Inject a Publisher while specifying the owning session

```
@Inject
@Name("orders")
@SessionName("Customers")
private Publisher<Order> orders;
```

3.6.5.3. Injecting a NamedTopic Subscriber

If application code only needs to subscribe to messages from a Coherence `NamedTopic` then instead of injecting a `NamedTopic` bean, a `Subscriber` bean can be injected.

The simplest way to inject a `Subscriber` is just to annotate the injection point of type `Subscriber` with `@Inject`, for example:

Inject Subscriber

```
@Inject
private Subscriber<Order> orders;
```

The example above will inject a `Subscriber` bean, the name of the underlying `NamedTopic` will be taken from the name of the injection point, in this case `orders`.

Specify the Topic Name

If the name of the injection point cannot be used as the `NamedTopic` name, which is always the case with injection points that are method or constructor parameters, then the `@Name` annotation can be used to specify the topic name.

For example, both of the code snippets below inject a `Subscriber` that subscribe to the `orders` topic:

Inject subscriber into field

```
@Inject
@Name("orders")
private Subscriber<Order> orders;
```

Inject subscriber into method parameter

```
@Controller
public class OrderController {
    @Inject
    public OrderController(@Name("orders") Subscriber<Order> topic) {
        // ToDo:
    }
}
```

Specify the Owning Session

As with injection of `NamedTopics`, in applications using multiple `Session` instances, the name of the `Session` that owns the underlying `NamedTopic` can be specified when injecting a `Subscriber` by adding the @SessionName annotation.

```
@Inject
@Name("orders")
@SessionName("Customers")
private Subscriber<Order> orders;
```

3.7. 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.

By default, the handling of Coherence events is asynchronous. Use the `@Synchronous` annotation to make the event handler execution synchronous.

Example of making a Coherence Event Listener synchronous

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

3.7.1. 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.

3.7.1.1. 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`.

3.7.1.2. 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.

3.7.1.3. 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`.

3.7.1.4. 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.

3.7.2. 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.

3.7.2.1. 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.

3.7.2.2. 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
}
```

3.7.2.3. 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
}
```

3.7.2.4. 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
}
```

3.7.2.5. 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
}
```

3.7.2.6. Entry Events

An `EntryEvent` 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 `EntryEvent`.

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

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

Receive Specific EntryEvent Types

There are a number of different `EntryEvent` 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
- **Removing** - an entry is being deleted from a cache, use the `@Removing` annotation
- **Removed** - an entry has been deleted from a cache, use the `@Removed` annotation

To restrict the `EntryEvent` 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 `Removed` events.

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

The event types fall into two categories, pre-events (those named **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
}
```

3.7.2.7. 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
}
```

3.7.2.8. 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
}
```

3.7.2.9. 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 therefore 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
}
```

3.7.2.10. 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
}
```

3.8. 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 example, 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`.

3.8.1. 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.

3.8.2. 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

3.8.3. 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
@CoherenceCache("plants")                                ③
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 Due to Spring limitations regarding the injection of Maps, we use the `@CoherenceMap` annotation to inject the `NamedMap`, which also has takes an optional value to specify the name of the cache.

3.9. 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.

3.9.1. 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.

3.9.2. 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.

3.9.3. 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
    @CoherenceMap("plants")                                  ③
    private NamedMap<Long, String> plants;                   ④
```

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 Due to Spring limitations regarding the injection of Maps, we use the `@CoherenceMap` annotation to inject the `NamedMap`, which also has takes an optional value to specify the underlying cache/map name to use for the view.

④ 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.

3.10. Messaging with Coherence Topics

Spring Coherence integration provides support for message driven applications by virtue of Coherence topics.

A Coherence NamedTopic is analogous to a queue or pub/sub topic, depending on the configuration and application code. Messages published to the topic are stored in Coherence caches, so topics are scalable and performant.

A typical stand-alone Coherence application would create a `NamedTopic` along with `Publisher` or `Subscriber` instances to publish to or subscribe to topics. Injection of topics into Spring applications is already covered in Injecting NamedTopics. With Spring messaging this becomes much simpler.

With Spring Coherence Messaging publishers and subscribers beans are created by writing suitably annotated interfaces.

3.10.1. Define Publishers - @CoherencePublisher

To create a topic Publisher that sends messages, you can simply define an interface that is annotated with @CoherencePublisher. Also, your configuration class has to be annotated with the @CoherencePublisherScan annotation. This is needed to specify the base package from which we recursively scan for `@CoherencePublisher` annotated interfaces.

Config.java

```
@Configuration
@CoherencePublisherScan("com.example.app.services")
public class Config {
}
```

For example the following is a trivial `@CoherencePublisher` interface:

ProductClient.java

```
import com.oracle.coherence.spring.annotation.CoherencePublisher;
import com.oracle.coherence.spring.annotation.Topic;

@CoherencePublisher  ①
public interface ProductClient {

    @Topic("my-products") ②
    void sendProduct(String message); ③

    void sendProduct(@Topic String topic, String message); ④
}
```

① The `@CoherencePublisher` annotation is used to designate this interface as a message publisher.

② The @Topic annotation indicates which topics the message should be published to

③ The method defines a single parameter, which is the message value. In this case the values being published are String instances but they could be any type that can be serialized by Coherence.

④ It is also possible for the topic to be dynamic by making it a method argument annotated with `@Topic`.

At run time Spring will produce an implementation of the above interface. You can retrieve an instance of `ProductClient` either by looking up the bean from the `ApplicationContext` or by injecting the bean with `@Inject`:

Using ProductClient

```
ProductClient client = applicationContext.getBean(ProductClient.class);
client.sendProduct("Blue Trainers");
```

3.10.2. Reactive and Non-Blocking Method Definitions

The `@CoherencePublisher` annotation supports the definition of reactive return types (such as Reactor `Flux`) as well as Futures.

The following sections cover possible method signatures and behaviour:

3.10.2.1. Mono Value and Return Type

```
Mono<Publisher.Status> sendBook(Mono<Book> book);
```

The implementation will return a Mono that when subscribed to will subscribe to the passed Mono and send a message emitting the resulting Publisher.Status.

3.10.2.2. Reactor Flux Value and Return Type

```
Flux<Publisher.Status> sendBooks(Flux<Book> book);
```

The implementation will return a Reactor Flux that when subscribed to will subscribe to the passed `Flux` and for each emitted item will send a message emitting the resulting Publisher.Status.

3.10.2.3. Future Return Type

```
Future<Publisher.Status> sendBooks(Mono<Book> book);
```

The implementation will return a Future with publisher's status.

3.10.3. Define Subscribers - @CoherenceTopicListener

To listen to Coherence topic messages you can use the @CoherenceTopicListener annotation to define a message listener.

The following example will listen for messages published by the `ProductClient` in the previous section:

ProductListener.java

```
import com.oracle.coherence.spring.annotation.CoherenceTopicListener;
import com.oracle.coherence.spring.annotation.Topic;

@CoherenceTopicListener   ①
public class ProductListener {

    @Topic("my-products")   ②
    public void receive(String product) { ③
        System.out.println("Got Product - " + product);
    }
}
```

① The @CoherenceTopicListener annotation to indicate that this bean is a Coherence topic listener.

② The @Topic annotation is again used to indicate which topic to subscribe to.

③ The receive method defines single arguments that will receive the message value, in this case the message is of type `String`.

3.10.4. Method Parameter Bindings

When using a Coherence topic Subscriber directly in application code, the `receive` method returns an Element, which contains the message value and metadata. The annotated subscriber method can take various parameter types that will bind to the element itself or to the message.

For example

```
@CoherenceTopicListener
@Topic("my-products")
public void receive(Element<Product> product) {
    // ... process message ...
}
```

The method above will be passed the Element received from the topic. By receiving the element, the method has access to the message value and all the metadata stored with the message.

3.10.5. Committing Messages

An important part of Coherence topic subscribers is committing messages to notify the server that they have been processed and guaranteeing at least once delivery. When using Micronaut Coherence messaging every message will be committed after the handler method has successfully processed the message. This behaviour can be controlled by adding a commit strategy to the @CoherenceTopicListener annotation.

3.10.5.1. Default Commit Behaviour

If no `commitStrategy` field has been provided to the `@CoherenceTopicListener` annotation the default behaviour is to synchronously call `Element.commit()` for every message received.

```
@CoherenceTopicListener
@Topic("my-products")
public void receive(Element<Product> product) {
    // ... process message ...
}
```

No `commitStrategy` field has been supplied to the `@CoherenceTopicListener` annotation.

3.10.5.2. Setting Commit Strategy

The @CoherenceTopicListener `commitStrategy` field is an enumeration of type CommitStrategy with three values, `SYNC`, `ASYNC` and `MANUAL`.

- CommitStrategy.SYNC - This strategy is the default, and will synchronously commit every message upon successful completion of the handler method, by calling `Element.commit()`.

```
@CoherenceTopicListener(commitStrategy = CommitStrategy.SYNC)
@Topic("my-products")
public void receive(Product product) {
    // ... process message ...
}
```

- CommitStrategy.ASYNC - This strategy will asynchronously commit every message upon successful completion of the handler method, by calling `Element.commitAsync()`.

```
@CoherenceTopicListener(commitStrategy = CommitStrategy.ASYNC)
@Topic("my-products")
public void receive(Product product) {
    // ... process message ...
}
```

- CommitStrategy.MANUAL - This strategy will not automatically commit messages, all handling of commits must be done as part of the handler method or by some external process.

```
@CoherenceTopicListener(commitStrategy = CommitStrategy.MANUAL)
@Topic("my-products")
public void receive(Element<Product> product) {
    // ... process message ...

    // manually commit the element
    element.commit();
}
```

In the example above a `MANUAL` commit strategy has used. The element will be committed by the application code at the end of the handler method. To be able to manually commit a message the method must take the `Element` as a parameter so that application code can access the commit

methods.

3.10.5.3. Forwarding Messages with @SendTo

On any @@CoherenceTopicListener method that returns a value, you can use the @SendTo annotation to forward the return value to the topic or topics specified by the @SendTo annotation.

The key of the original ConsumerRecord will be used as the key when forwarding the message.

ProductListener.java

```
import com.oracle.coherence.spring.*;
import org.springframework.messaging.handler.annotation.SendTo;

@CoherenceTopicListener
public class ProductListener {

    @Topic("awesome-products")      ①
    @SendTo("product-quantities")   ②
    public int receive(Product product) {
        System.out.println("Got Product - " + product.getName() + " by " +
product.getBrand());
        return product.getQuantity(); ③
    }
}
```

① The topic subscribed to is `awesome-products`

② The topic to send the result to is `product-quantities`

③ The return value is used to indicate the value to forward

You can also do the same using Reactive programming:

ProductListener.java

```
import com.oracle.coherence.spring.*;
import org.springframework.messaging.handler.annotation.SendTo;
import reactor.core.publisher.Mono;

@CoherenceTopicListener
public class ProductListener {

    @Topic("awesome-products")       ①
    @SendTo("product-quantities")    ②
    public Mono<Integer> receiveProduct(Mono<Product> productSingle) {
        return productSingle.map(product -> {
            System.out.println("Got Product - " + product.getName() + " by " +
product.getBrand());
            return product.getQuantity();  ③
        });
    }
}
```

① The topic subscribed to is `awesome-products`

② The topic to send the result to is `product-quantities`

③ The return is mapped from the single to the value of the quantity

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 following resources:

- The Cache Abstraction chapter in the core Spring Framework reference guide
- Spring Boot's reference documentation's support regarding caching

Spring's cache abstraction for Coherence will be automatically enabled as soon as you specify `@EnableCaching` in your applications configuration classes. In that case a `CoherenceCacheManager` bean implementation is registered as `CacheManager`. Of course, you can define your own `CacheManager` as well, but in that case auto-configuration will back-off.

The autoconfiguration logic is defined in class EnableCoherenceImportBeanDefinitionRegistrar.

Example 4. Defining your own CacheManager

Java

```
@Configuration
@EnableCaching
public class CacheConfiguration {
    @Bean
    public CacheManager cacheManager(Coherence coherence) {
      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="cacheManager"
class="com.oracle.coherence.spring.cache.CoherenceCacheManager">
    <constructor-arg ref="coherenceInstance"/>
  </bean>
</beans>
```

Please consult the Quickstart chapter to see an example using Spring's cache abstraction support with Spring Boot. Coherence Spring also provides an example of using Spring Framework (without Spring Boot). The source code for the samples is part of the Coherence Spring projects:

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

If you're using Spring Boot, please continue reading the Spring Boot specific chapter on caching.

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

In this chapter you will learn how to configure Coherence as an HTTP session store using Spring Session.

First you need to add the coherence-spring-session dependency:

Example 5. Adding the Coherence Spring Session Dependency

Maven

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

Gradle

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

Coherence Spring support for Spring Session can be used for either the free Coherence Community Edition (CE) or the commercial version. Coherence Spring does not bring in the Coherence dependency automatically but users must specify the Coherence dependency explicitly.

In order to enable Spring Session support for Coherence, make sure Coherence is enabled and then enable Spring Session using the `@EnableCoherenceHttpSession` annotation.

Enabling Coherence Spring Sessions

```
@Configuration
@EnableCoherence
@EnableCoherenceHttpSession(                    ①
        session = "coherence_session",          ②
        cache = "spring:session:sessions",      ③
        flushMode = FlushMode.ON_SAVE,          ④
        sessionTimeoutInSeconds = 1800          ⑤
)
static class CoherenceSessionConfig {
}
```

① Enables Spring Session support for Coherence

② Specify the name of the Coherence Session. Optional. Defaults to Coherence' default session.

③ The name of the cache to use. Optional. Defaults to `spring:session:sessions`.

④ The FlushMode to use. Optional. Defaults to `FlushMode.ON_SAVE`.

⑤ The session timeout. Optional. Defaults to `1800` seconds (`30` minutes)

Are you running Coherence as a dedicated server instance? Then you need to make sure that your Coherence server may need one or more additional dependencies on its classpath for serialization. Depending on your requirements, you may need `Coherence Spring Session`, `Spring Security Core`, `Spring Security Web`. Please also ensure that dependency version between Coherence server and application clients matches exactly.

5.2. POF Serialization

In case that you configured the cache using POF serialization, additional POF configuration for the class `MapSession` is necessary:

POF Configuration

```
 <user-type>
     <type-id>2001</type-id>
     <class-name>org.springframework.session.MapSession</class-name>
     <serializer>
         <class-
name>com.oracle.coherence.spring.session.serialization.pof.MapSessionPofSerializer</cl
ass-name>
     </serializer>
 </user-type>
```

Depending on your serialization requirements and your session data, additional POF configuration may be necessary.

5.3. Spring Session Sample

This Coherence Spring source code repository provides a dedicated example application, illustrating the usage of Spring Session and Spring Security using Spring Boot.

The example application show-cases 2 use-cases:

- Use Spring Session with embedded Coherence instances and Java serialization
- Use Spring Session with remote Coherence instances (Coherence*Extends) and Java serialization

Even though this demo is targeting Spring Session, we use Spring Security as well, since the authentication details are stored in the session as well. In regard to authentication, users have 2 choices:

- A user can POST a JSON request containing the username and password in the body of the request.
- Use basic authentication

The username is `coherence`, and the password `rocks`.

Example Authentication Request

```
{ "username": "coherence", "password": "rocks" }
```

Once successfully authenticated, the application will return a `2xx` HTTP status with an empty body and a session cookie. An Authentication failure, on the other hand, produces a non-2xx HTTP status with an empty body. The application has an endpoint that responds to a `GET` request to the URL `/hello` that returns the string `Hello Coherence`. All endpoints require an authenticated user using the session cookie or the `username` and `password`.

5.3.1. Start Spring Session with Embedded Coherence Instances

Build the Coherence Server instance

```
$ ./mvnw clean package -pl samples/spring-session-demo/spring-session-demo-app
```

Now we are ready to run the application. Let's launch 2 instances, one listening on the pre-configured port `8090`, and the other one on port `8091`.

Run the Spring Boot application

```
 $ java -jar samples/spring-session-demo/spring-session-demo-app/target/spring-
session-demo-app-3.0.0-SNAPSHOT.jar
 $ java -jar samples/spring-session-demo/spring-session-demo-app/target/spring-
session-demo-app-3.0.0-SNAPSHOT.jar --server.port=8091
```

5.3.2. Spring Session with Remote Coherence Instances

Build the Coherence Server instance

```
$ ./mvnw clean package -pl samples/spring-session-demo/spring-session-demo-server
```

Build the Application instance

```
$ ./mvnw clean package -pl samples/spring-session-demo/spring-session-demo-app
```

Now we are ready to run the application. We will activate the `coherence-client` Spring Boot profile as well:

Run the Spring Boot application

```
 $ java -jar samples/spring-session-demo/spring-session-demo-app/target/spring-
session-demo-server-3.0.0-SNAPSHOT.jar
 $ java -jar samples/spring-session-demo/spring-session-demo-app/target/spring-
session-demo-app-3.0.0-SNAPSHOT.jar
```

5.3.3. Accessing the REST Endpoints

Log into the application using CURL

```
$ curl -i -c cookie.txt \
-H "Accept: application/json" \
-H "Content-Type:application/json" \
-X POST --data '{"username": "coherence", "password": "rocks"}' \
"http://localhost:8090/login"
```

Let's access the HelloController

```
$ curl -i -b cookie.txt \
-H "Accept: application/json" \
-H "Content-Type:application/json" \
-X GET "http://localhost:8090/hello"
```

5.3.4. Spring Session Actuator

Using Spring Boot's Actuator endpoints, we can introspect the session using the Sessions actuator at `localhost:8090/actuator/sessions?username=coherence`.

Retrieving session information for user `coherence`

```
$ curl -i -b cookie.txt \
-H "Accept: application/json" \
-H "Content-Type:application/json" \
-X GET "http://localhost:8090/actuator/sessions?username=coherence"
```

5.3.5. Generate Docker Image

If you prefer to use Docker, you can create an image using:

Generate a Docker image

```
 $ mvn spring-boot:build-image -pl samples/spring-session-demo/spring-session-demo-app
-Dspring-boot.build-image.imageName=coherence/spring_session_demo
```

5.4. Session Expiration Strategies

When dealing with the expiration of cache entries, you generally have 2 options in Coherence:

- Set the expiration time for each put operation explicitly
- Configure caches in your `coherence-cache-config.xml` file

When you define a session timeout via the application, for example `@EnableCoherenceHttpSession(sessionTimeoutInSeconds = 1000)`, the session expiration will be set for each put-operation in `CoherenceIndexedSessionRepository`.

If not set in the application, Coherence Spring will expire HTTP session caches in `1800` seconds (`30` minutes).

If you rather prefer defining the session expiration timeouts in your `coherence-cache-config.xml` file, you should set the session timeout in the application to `0`, for instance `@EnableCoherenceHttpSession(sessionTimeoutInSeconds = 0)`. That way, put operations will never to set an expiration value for the cache entry. You can then set the `expiry-delay` cache configuration element for your cache in the `coherence-cache-config.xml` file.

In regard to the question, whether one strategy or the other strategy is preferable: It is mostly a matter of preference. You do have, however, a bit more control when configuring expiration logic via the `coherence-cache-config.xml` file, as you have the ability to define custom eviction policies.

For more information, please consult the respective chapter on *Controlling the Growth of a Local Cache* in the Coherence reference guide.

The underlying expiry delay parameter in Coherence is defined as an integer and is expressed in milliseconds. Therefore, the maximum amount of time can never exceed Integer.MAX_VALUE (2147483647) milliseconds or approximately 24 days.

Chapter 6. Coherence Spring Data

6.1. Introduction

The Spring Data Coherence module provides integration with Coherence data grids. Key functional areas of Spring Data Coherence are a POJO centric model for interacting with a Coherence data grid and easily writing a Repository style data access layer.

6.2. Features

- Spring configuration support using Java-based @Configuration classes.
- Automatic implementation of Repository interfaces
- Rich query and event features from Coherence
- Native asynchronous repository support
- Projections

6.3. Getting Started

Example 6. Coherence Spring Data Dependencies

Maven

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

Gradle

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

6.4. Defining Repositories

Before proceeding, please be familiar with the Spring Data Commons documentation, as this section will assume some familiarity with Spring Data.

Simple repositories such as the following will, of course, work as expected:

```
public interface PersonRepository extends CrudRepository<String, Person> {
    // ...
}
```

However, it is recommended to extend the CoherenceRepository interface, to fully utilize the features Coherence for Spring Data has to offer such as:

- Powerful projection features
- Flexible in-place entity updates
- First-class data aggregation support
- Stream API support
- Event listener support
- Declarative acceleration and index creation
- Native asynchronous support

Please also see the Coherence Repository documentation for more details on these features.

Example extending the blocking `CoherenceRepository` interface:

```
import com.oracle.coherence.spring.data.repository.CoherenceRespository;

// ...

public interface PersonRepository extends CoherenceRepository<String, Person> {
    // ...
}
```

or for the non-blocking flavor:

```
import com.oracle.coherence.spring.data.repository.CoherenceAsyncRespository;

// ...

public interface PersonRepository extends CoherenceAsyncRepository<String, Person> {
    // ...
}
```

6.4.1. Identifying the Coherence NamedMap

The Coherence `NamedMap` that will be used by the Repository implementation will be based on the type name in the Repository class assuming the Repository name follows the format of [Type]Repository (e.g., PersonRepository will use a NamedMap called `person`). If this is not desired, the name may instead be passed by the `@CoherenceMap` annotation. For example:

```
import com.oracle.coherence.spring.data.config.CoherenceMap;
import com.oracle.coherence.spring.data.repository.CoherenceRespository;

// ...

@CoherenceMap("people")
public interface PersonRepository extends CoherenceRepository<String, Person> {
    // ...
}
```

6.5. Mapping Entities

As Coherence is, at its core, a key-value store, mapping Entities for use with a Coherence Repository is relatively simple as only the id needs to be annotated. It is possible to use either `org.springframework.data.annotation.Id` or `javax.persistence.Id` to denote the entity's id.

For example:

```
public class Person implements Serializable {
    @org.springframework.data.annotation.Id
    protected String id;

    // ---- person functionality ----
}
```

6.6. Using the Repository

In order to enable Coherence-based Repositories, you must use the `@EnableCoherenceRepositories` annotation. A simple configuration example would be:

```
import com.oracle.coherence.spring.configuration.annotation.EnableCoherence;
import
com.oracle.coherence.spring.configuration.data.config.EnableCoherenceRepositories;

// ...

@Configuration
@EnableCoherence
@EnableCoherenceRepositories
public static class Config {
}
```

Similarly to other Spring Data implementations, the `@EnableCoherenceRepositories` annotation offers several configuration options to configure how Spring will search for repositories. Please see the API docs for details.

6.6.1. Finder Queries

One of the benefits of Spring Data is the ability to define queries on the Repository interface using Spring Data's finder query syntax. For example:

```
import com.oracle.coherence.spring.data.repository.CoherenceRespository;
import com.oracle.coherence.spring.data.model.Author;
import com.oracle.coherence.spring.data.model.Book;
import com.tangosol.util.UUID;

// ...

public interface BookRepository extends CoherenceRepository<Book, UUID> {
    List<Book> findByAuthor(Author author);
    // other finders
}
```

It should be noted that finder queries defined on either the blocking or non-blocking Coherence repository will always execute in a blocking manner. For asynchronous versions of these methods, please use Spring's Async Method support.

```
import com.oracle.coherence.spring.data.config.CoherenceMap;
import com.oracle.coherence.spring.data.model.Author;
import com.oracle.coherence.spring.data.model.Book;
import com.oracle.coherence.spring.data.repository.CoherenceAsyncRepository;
import com.tangosol.util.UUID;

import org.springframework.scheduling.annotation.Async;

// ...

@CoherenceMap("book")
public interface CoherenceBookAsyncRepository extends CoherenceAsyncRepository<Book,
UUID> {

    @Async
    CompletableFuture<List<Book>> findByAuthor(Author author);
}
```

Don't forget to include the `@EnableAsync` annotation on the configuration:

```
import com.oracle.coherence.spring.configuration.annotation.EnableCoherence;
import
com.oracle.coherence.spring.configuration.data.config.EnableCoherenceRepositories;
import org.springframework.scheduling.annotation.EnableAsync;

// ...

@Configuration
@EnableAsync
@EnableCoherence
@EnableCoherenceRepositories
public static class Config {
}
```

6.7. Projections

Spring Data Coherence module supports projections as defined in Spring Data Projections documentation. This allows us, among other things, to transfer a subset of the entities properties when `closed` projections are used.

Imagine a repository and aggregate root type such as the following example:

```
@Entity
public class Book implements Cloneable, Serializable {
    @Id
    protected final UUID uuid;
    protected String title;
    protected Author author;
    protected int pages;
    protected Calendar published;
}

@CoherenceMap("book")
interface BookRepository extends CrudRepository<Book, UUID> {

    List<BookProjection> findByTitle(String title);

    // ...
}
```

6.7.1. Interface-based Projections

The simplest way to simplify the result is to declare an interface that exposes methods reading the desired properties, as shown in the following example:

A projection interface to retrieve a subset of attributes

```
interface BookTitleAndPages {

    String getTitle();
    int getPages();
}
```

A repository using an interface based projection with a query method

```
interface BookRepository extends CrudRepository<Book, UUID> {

    List<BookTitleAndPages> findByTitle(String title);
}
```

The query execution engine creates proxy instances of that interface at runtime for each element returned and forwards calls to the exposed methods to the target object.

Projections can be used recursively as shown in the following example:

A projection interface to retrieve a subset of attributes

```
interface BookSummary {

    String getTitle();
    int getPages();
    AuthorSummary getAuthor();

    interface AuthorSummary {
        String getFirstName();
    }
}
```

6.7.2. Closed Projections

A projection interface whose accessor methods all match properties of the target aggregate is considered to be a closed projection. The following example is a closed projection:

A closed projection

```
interface BookTitleAndPages {

    String getTitle();
    int getPages();
}
```

6.7.3. Open Projections

Accessor methods in projection interfaces can also be used to compute new values by using the @Value annotation, as shown in the following example:

An open projection

```
interface BookTitleAndAuthor {

    @Value("#{target.author.firstName + ' - ' + target.title}")
    String getTitleAndAuthor();
}
```

A projection interface using `@Value` is an open projection. Spring Data cannot apply query execution optimizations in this case, because the SpEL expression could use any attribute of the aggregate root.

Default methods also can be used for open projection interfaces:

A projection interface using a default method for custom logic

```
interface BookTitleAndAuthor {

    default String getTitleAndAuthor() {
        return getAuthor().getFirstName().concat(' - ').concat(getTitle());
    }
}
```

A more flexible option is to implement the custom logic in a Spring bean:

A projection interface using a default method for custom logic

```
@Component
class MyBean {

    String getTitleAndAuthor(Book book) {
        return book.getAuthor().getFirstName().concat(' - ').concat(book.getTitle());
    }
}

interface BookTitleAndAuthor {

    @Value("#{@myBean.getTitleAndAuthor(target)}")
    String getTitleAndAuthor();
}
```

Methods backed by SpEL expression evaluation can also use method parameters, which can then be referred to from the expression. The method parameters are available through an Object array named args. The following example shows how to get a method parameter from the args array:

A sample Book object

```
interface NameOnly {

    @Value("args[0] + ' ' + #{target.author.firstName}")
    String getHonorificName(String honorific);
}
```

6.7.4. Nullable Wrappers

Getters in projection interfaces can make use of nullable wrappers for improved null-safety. Currently supported wrapper types are:

- java.util.Optional
- com.google.common.base.Optional
- scala.Option
- io.vavr.control.Option

A projection interface using nullable wrappers

```
interface TitleOnly {

    Optional<String> getTitle();
}
```

6.7.5. Class-based Projections (DTOs)

Another way of defining projections is by using value type DTOs (Data Transfer Objects) that hold properties for the fields that are supposed to be retrieved - similar to the projection interfaces except that no proxying happens and no nested projections can be applied.

The fields to be loaded are determined from the parameter names of the constructor that is exposed.

The following example shows a projecting DTO:

A projecting DTO

```
class BookTitleAndPages {
    private final String title;
    private final int pages;

    BookTitleAndPages(String title, int pages) {
        this.title = title;
        this.pages = pages;
    }

    String getTitle() {
        return this.title;
    }

    int getPages() {
        return this.pages;
    }

    // equals(…) and hashCode() implementations
}
```

6.7.6. Dynamic Projections

It's possible to select the return type to be used at invocation time (up to now all examples shown exact projection type to be used as a return type). To apply dynamic projections, use a query method such as the one shown in the following example:

A repository using a dynamic projection parameter

```
@CoherenceMap("book")
interface BookRepository extends CrudRepository<Book, UUID> {

    <T> Collection<T> findByTitle(String title, Class<T> type);
}
```

Using a repository with dynamic projections

```
Collection<BookSummary> aggregates = books.findByTitle("Shadow", BookSummary.class);

Collection<BookTitleAndPages> aggregates = books.findByTitle("Remember",
BookTitleAndPages.class);
```

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.

Example 7. Adding the Coherence Spring Boot Starter Dependency

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</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")
}
```

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 class for further configuration. The YAML configuration for Spring Boot's Coherence support may look like the following:

```
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	`slf4j`	The type of the logging destination.
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`	Specifies a logger name within chosen logging system
coherence.logging.message-format		Specifies how messages that have a logging level specified are formatted before passing them to the log destination.
coherence.logging.character-limit		Specifies the maximum number of characters that the logger daemon processes from the message queue before discarding all remaining messages in the queue
coherence.sessions.<type>	N/A	Indicates the type of the session. One of the following: `grpc`, `server`, `client`

Key	Default Value	Description
coherence.sessions.<type>[0].name		Name of the session.
coherence.sessions.<type>[0].config		The Coherence cache configuration URI for the session
coherence.sessions.<type>[0].priority	`SessionConfiguration#DEFAULT_PRIORITY`	The priority order to be used when starting the session. Sessions will be started with the lowest priority first.
coherence.sessions.<type>[0].scope-name		The scope name for the session.

`Session`-related configuration properties are defined based on the session type in:

- AbstractSessionConfigurationBean
- SessionConfigurationBean
- GrpcSessionConfigurationBean

All but the `Session` properties 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.

If the provided Spring Boot configuration properties are not sufficient for your needs, you can also specify any of the native Coherence properties. For a list of available native properties, please consult the reference guide chapter on System Property Overrides.

7.3. Coherence Support of the Spring Boot ConfigData API

Since Spring Boot `2.4.x`, you can define your own custom config locations. This allows you to import these config locations 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.

You can trigger the import of remote configuration properties with the Spring Boot property `spring.config.import` using the value `coherence:`. You will also need the corresponding Coherence config-client configured to specify any remote connection settings for your Coherence cluster as well as settings specifying how properties will be applied, e.g., the name of the application used to fetch remote properties.

```
spring:
  config:
    import: "coherence:"
coherence:
  config-client:
    application-name: berlin
    profile: kona
```

Many properties have sensible default value. Please see class CoherenceConfigClientProperties for details.

7.4. Caching with Spring Boot

If you have not already done so, please read the chapter Configuring Coherence Cache for Spring first.

In this chapter, we see how we can use caching with Coherence in Spring Boot. As long as `coherence-spring-boot-starter` is added to your project, and caching is enabled via `@EnableCaching`, Coherence Autoconfiguration will automatically provide a CoherenceCacheManager (implementing Spring's CacheManager) to the ApplicationContext. However, the `CacheManager` is only added if no `CacheManager` was configured explicitly beforehand.

Once that is in place, you can take advantage of Spring's Cache abstraction backed by Coherence and use Spring's annotations such as `@Cacheable`, `@CacheEvict`, `@CachePut` etc.

When learning about Coherence's Spring cache abstraction support, please familiarize yourself with the following resources:

- Coherence Spring Quickstart chapter
- Coherence Core Spring Cache Support
- Spring Boot's reference documentation's support regarding caching
- The Cache Abstraction chapter in the core Spring Framework reference guide

7.5. Configure Hibernate Second-Level Caching

Spring's cache abstraction is not your only option with regard to caching. Another option is to use Hibernate's Second-Level cache support.

Currently, Coherence only supports Hibernate up to `5.2.x`, as the Hibernate Cache SPI changed with Hibernate `5.3.x` and higher. Please track the relevant GitHub issue for the Coherence Hibernate project.

When using only Hibernate's second-level cache without the requirement of using Coherence for non-Hibernate application needs, you may want to disable Coherence's auto-configuration support for Spring Boot as it is not needed:

Disable CoherenceAutoConfiguration

```
@SpringBootApplication(exclude=CoherenceAutoConfiguration.class)
public class HibernateApplication {
    public static void main(String[] args) {
        SpringApplication.run(HibernateApplication.class, args);
    }
}
```

In order to configure Hibernate Second-Level Caching for Coherence using Spring Boot, you simply have to set the Hibernate property `hibernate.cache.region.factory_class` to `com.oracle.coherence.hibernate.cache.CoherenceRegionFactory`.

Your Spring Boot `application.yaml` file may look like the following:

RegionFactory Configuration using YAML

```
spring:
  application:
    name: Your Spring Boot App
  jpa:
    properties:
      hibernate.cache.region.factory_class:
com.oracle.coherence.hibernate.cache.CoherenceRegionFactory ①
      hibernate.cache.use_query_cache: true
      hibernate.cache.use_second_level_cache: true
      hibernate.format_sql: true
      hibernate.generate_statistics: true
      hibernate.show_sql: true
      hibernate.use_sql_comments: false
...
```

① Configuring the `CoherenceRegionFactory`

For more detailed information please see the Caching chapter of the Hibernate reference documentation.

7.5.1. Hibernate Second Level Cache Example

The Coherence Spring source code repository also provides an example application, illustrating the Coherence support for Hibernate's Second Level caching using Spring Boot.

The example application has 1 REST endpoint that can persist, retrieve and delete a `Plant` object with taxonomic properties such as the genus and species.

Example of a `Plant` JSON response

```
{"id": 1, "genus": "Sabal", "species": "minor", "commonName": "Dwarf palmetto"}
```

7.5.1.1. Run the Hibernate Application

First, let's build the application:

Build the Coherence Server instance

```
 $ ./mvnw clean package -pl samples/hibernate-cache-demo
```

Next, run the application:

Run the Spring Boot application

```
 $ java -jar samples/hibernate-cache-demo/target/hibernate-cache-demo-3.0.0-
SNAPSHOT.jar
```

The application will be available on port `9090`.

Add a plant

```
curl -i \
  -H "Accept: application/json" \
  -H "Content-Type:application/json" \
  -X POST --data '{"genus": "Sabal", "species": "minor", "commonName": "Dwarf
palmetto"}' \
  "http://localhost:9090/plants"
```

List all plants

```
curl -i \
  -H "Accept: application/json" \
  -H "Content-Type:application/json" \
  -X GET "http://localhost:9090/plants"
```

Get a single plant

```
curl -i \
  -H "Accept: application/json" \
  -H "Content-Type:application/json" \
  -X GET "http://localhost:9090/plants/1"
```

Delete a plant

```
curl -i \
  -H "Accept: application/json" \
  -H "Content-Type:application/json" \
  -X DELETE "http://localhost:9090/plants/1"
```

When retrieving a single plant repeatedly, you will see no SQL printed to the console. Instead, the

plant is returned from the second level cache as illustrated by the printed out cache statistics.

Hibernate Cache Statistics

```
2021-06-28 16:12:42.545  INFO 29022 --- [nio-9090-exec-7]
i.StatisticalLoggingSessionEventListener : Session Metrics {
27579 nanoseconds spent acquiring 1 JDBC connections;
0 nanoseconds spent releasing 0 JDBC connections;
0 nanoseconds spent preparing 0 JDBC statements;
0 nanoseconds spent executing 0 JDBC statements;
0 nanoseconds spent executing 0 JDBC batches;
0 nanoseconds spent performing 0 L2C puts;
1587296 nanoseconds spent performing 1 L2C hits;
0 nanoseconds spent performing 0 L2C misses;
0 nanoseconds spent executing 0 flushes (flushing a total of 0 entities and 0
collections);
0 nanoseconds spent executing 0 partial-flushes (flushing a total of 0 entities and 0
collections)
}
```

In order to also cache the results of query executions, you must provide query hints to Hibernate. The `PlantRepository` class has a custom query using such a hint for illustration purposes:

Using Query Hints

```
@QueryHints(@QueryHint(name = org.hibernate.annotations.QueryHints.CACHEABLE, value =
"true"))
@Query("select p from Plant p")
List<Plant> getAllPlantsCacheable();
```

You can see the cache being used by retrieving the list of plants with an additional request parameter `use-cache`:

List all plants

```
curl -i \
  -H "Accept: application/json" \
  -H "Content-Type:application/json" \
  -X GET "http://localhost:9090/plants?use-cache=true"
```

7.6. Spring Session Support

If you have not already done so, please read the chapter on Coherence Spring Session first.

Coherence Spring also provides support for Spring Session. This includes auto-configuration support as well as a set of Coherence-specific configuration properties.

The following Coherence-specific configuration properties are available:

Table 2. Coherence Configuration Properties

Key	Default Value	Description
coherence.spring.session.enabled	`false`	Set to `true` to enable Spring Session support
coherence.spring.session.map-name	`spring:session:sessions`	Name of the map used to store sessions
coherence.spring.session.flush-mode	`ON_SAVE`	Sessions flush mode. Determines when session changes are written to the session store.
coherence.spring.session.save-mode	`ON_SET_ATTRIBUTE`	The session save mode determines how session changes are tracked and saved to the session store.

7.7. Coherence Messaging Support

Support for injecting Coherence Publishers is enabled automatically by adding the `coherence-spring-boot-starter` dependency. By default, base package to scan for `@CoherencePublisher` annotated interfaces is derived from the main class annotated with `@SpringBootApplication`. Base package can be overridden by annotating configuration class with `@CoherencePublisherScan` as described at Messaging with Coherence Topics.

7.8. Coherence Metrics

By adding the `coherence-spring-boot-starter` and `com.oracle.coherence.ce:coherence-micrometer` dependencies auto-configuration will register `CoherenceMicrometerMetrics` and publish Coherence Metrics under the `coherence.` name.

To disable the auto-configured Coherence metrics even when `coherence-micrometer` module is on the classpath, set the following property:

application.yaml

```
management:
  metrics:
    enable:
      coherence: false
```

Chapter 8. Coherence Spring Cloud Config

This section explains how to configure Coherence using Spring Cloud Config.

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

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.

Set up 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.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 as well.

For instance the following properties are equivalent:

Equivalent Properties

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

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

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

Chapter 9. Appendices