Services and Registries

Table of Contents

* [What is a Service?](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_what_is_a_service)
* [What is a ServiceRegistry?](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_what_is_a_serviceregistry)
* [ServiceBinding](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_servicebinding)
* [Types of ServiceRegistries](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_types_of_serviceregistries)
  + [BootstrapServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_bootstrapserviceregistry)
  + [StandardServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_standardserviceregistry)
  + [SessionFactoryServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_sessionfactoryserviceregistry)
* [Service lifecycle](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_service_lifecycle)
  + [Initiation (creation)](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-initiation)
  + [Configuration](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-configuration)
  + [Starting](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-starting)
  + [Stopping](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-stopping)
  + [Manageable (JMX)](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_manageable_jmx)
* [Service Dependencies](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_service_dependencies)
* [Building ServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#serviceregistry-building)
  + [Building BootstrapServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_building_bootstrapserviceregistry)
  + [Building StandardServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_building_standardserviceregistry)
* [Conclusion](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#_conclusion)

Services and Registries are new **as a formalized concept** starting in 4.0. But the functionality provided by the different Services have actually been around in Hibernate much, much longer. What is new is managing them, their lifecycles and dependencies through a lightweight, dedicated container we call a ServiceRegistry. The goal of this guide is to describe the design and purpose of these Services and Registries, as well as to look at details of their implementations where appropriate. It will also delve into the ways third-party integrators and applications can leverage and customize Services and Registries.

What is a Service?

Services provide various types of functionality, in a pluggable manner. Specifically they are interfaces defining certain functionality and then implementations of those service contract interfaces. The interface is known as the service role; the implementation class is known as the service implementation. The pluggability comes from the fact that the service implementation adheres to contract defined by the interface of the service role and that consumers of the service program to the service role, not the implementation.

|  |  |
| --- | --- |
|  | All Services are expected to implement the org.hibernate.service.Service "marker" interface. Hibernate uses this internally for some basic type safety; it defines no methods (at the moment). |

Let’s look at an example to better define what a Service is. Hibernate needs to be able to access JDBC Connections to the database. The way it obtains and releases these Connections is through the ConnectionProvider service. The service is defined by the interface (service role) org.hibernate.engine.jdbc.connections.spi.ConnectionProvider which declares methods for obtaining and releasing the Connections. There are then multiple implementations of that service contract, varying in how they actually manage the Connections:

* org.hibernate.engine.jdbc.connections.internal.DatasourceConnectionProviderImpl for using a javax.sql.DataSource
* org.hibernate.c3p0.internal.C3P0ConnectionProvider for using a C3P0 Connection pool
* etc.

Internally Hibernate always references org.hibernate.engine.jdbc.connections.spi.ConnectionProvider rather than specific implementations in consuming the service (we will get to producing the service later when we talk about registries). Because of that fact, other ConnectionProvider service implementations could be plugged in.

There is nothing revolutionary here; programming to interfaces is generally accepted as good programming practice. What’s interesting is the ServiceRegistry and the pluggable swapping of the different implementors.

What is a ServiceRegistry?

A ServiceRegistry, at its most basic, hosts and manages Services. Its contract is defined by theorg.hibernate.service.ServiceRegistry interface.

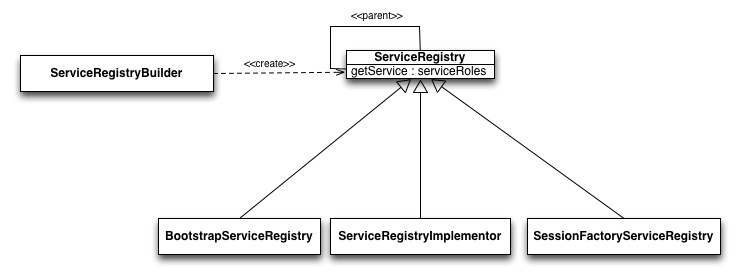
We already gave a basic overview and definition of services. But services have other interesting characteristics as well. Services have a lifecycle. They have a scope. Services might depend on other services. And they need to be produced (choose using one implementation over another). The ServiceRegistry fulfills all these needs.

In a concise definition, the ServiceRegistry acts as a inversion-of-control (IoC) container.

|  |  |
| --- | --- |
|  | Despite some recent revisionist history, Spring did not invent IoC nor dependency injection nor were they even the first to bring it into Java. Projects like JBoss MicroContainer and Apache Avalon pre-date Spring by many years and each did IoC and dependency injection. The concepts in ServiceRegistry are actually very similar to Apache Avalon. |

Why not just use an existing IoC framework? The main reason was that this had to be as light-weight and as small of a footprint as possible. The initial design also had called for Services to be swappable at runtime, which unfortunately had to be removed due to performance problems in the proxy-based solution to swapping (the plan is to investigate alternate ways to achieve swap-ability with better performance at a later date).

A Service is associated with a ServiceRegistry. The ServiceRegistry scopes the Service. The ServiceRegistry manages the lifecycle of the Service. The ServiceRegistry handles injecting dependencies into the Service (actually both a pull and a push/injection approach are supported). ServiceRegistries are also hierarchical, meaning a ServiceRegistry can have a parent ServiceRegistry. Services in one registry can depend on and utilize services in that same registry as well as any parent registries.



ServiceBinding

The association of a given Service to a given ServiceRegistry is called a binding and is represented by theorg.hibernate.service.spi.ServiceBinding interface. Furthermore, the specific contract between a ServiceBinding and the ServiceRegistry is represented by the org.hibernate.service.spi.ServiceBinding.ServiceLifecycleOwner interface.

There are 2 ways to associate a Service with a ServiceRegistry. The Service can be directly instantiated and then handed to the ServiceRegistry, or a ServiceInitiator can be given to the ServiceRegistry (which the ServiceRegistry will use if and when the Service is needed). ServiceRegistry implementations (those using the org.hibernate.service.internal.AbstractServiceRegistryImplconvenience base implementation) register bindings through calls to the overloaded AbstractServiceRegistryImpl#createServiceBinding method accepting either a Service instance or a ServiceInitiator instance. However, each specific ServiceRegistry type has a dedicated builder through which its Services are typically defined and customized.

Types of ServiceRegistries

Currently Hibernate utilizes 3 different ServiceRegistry implementations forming a hierarchy.

BootstrapServiceRegistry

The root ServiceRegistry is the org.hibernate.boot.registry.BootstrapServiceRegistry. BootstrapServiceRegistry is a specialization of org.hibernate.service.ServiceRegistry. The BootstrapServiceRegistry interface adds no new behavior, it is simply a specialization for the purpose of type safety. In normal usage, the BootstrapServiceRegistry has no parent.

|  |  |
| --- | --- |
|  | This registry holds services that absolutely have to be available for most things in Hibernate to work. |

The BootstrapServiceRegistry normally holds 3 services and is normally built by means of theorg.hibernate.boot.registry.BootstrapServiceRegistryBuilder class. The builder gives type safe access to customizing these 3 Services.

ClassLoaderService

This service exposes the capability to interact with ClassLoaders. The manner in which Hibernate (or any library) should interact with ClassLoaders varies based on the runtime environment which is hosting the application. Application servers, OSGi containers, and other modular class loading systems impose very specific class-loading requirements. This service is provides Hibernate an abstraction from this environmental complexity. And just as importantly, it does so in a centralized, swappable manner.

The specific capabilities exposed on this service include:

* Locating java.lang.Class references by name. This includes application classes as well as "integration" classes.
* Locating resources (properties files, xml files, etc) as "classpath resources"
* Interacting with java.util.ServiceLoader

The service role for this service is org.hibernate.boot.registry.classloading.spi.ClassLoaderService.

IntegratorService

Applications, third-party integrators and others all need to integrate with Hibernate. Historically this used to require something (usually the application) to coordinate registering the pieces of each integration needed on behalf of each integration. The org.hibernate.integrator.spi.Integrator formalized this "integration SPI". The IntegratorService manages all known integrators.

|  |  |
| --- | --- |
|  | The concept of "Integrator" is still being actively defined and developed as part of the 5.0 codebase. Expect changes in these SPIs; in fact those changes are already begun in the repository branch housing 5.0 development. |

There are 2 ways an integrator becomes known.

1. The integrator may be manually registered by calling BootstrapServiceRegistryBuilder#with(Integrator)
2. The integrator may be discovered, leveraging the standard Java java.util.ServiceLoader capability provided by the ClassLoaderService. Integrators would simply define a file named*/META-INF/services/org.hibernate.integrator.spi.Integrator* and make it available on the classpath.java.util.ServiceLoader covers the format of this file in detail, but essentially it lists classes by FQN that implement the org.hibernate.integrator.spi.Integrator one per line.

The service role for this service is org.hibernate.integrator.spi.IntegratorService.

StrategySelector

Think of this as the "short naming" service. Historically to configure Hibernate users would often need to give FQN references to internal Hibernate classes.

For example, to tell Hibernate to use JDBC-based transactions we need to tell it to use theorg.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory class by specifying its FQN as part of the config:

hibernate.transaction.factory\_class=org.hibernate.engine.transaction.internal.jdbc.JdbcTransactionFactory

Of course this has caused lots of problems as we refactor internal code and move these classes around into different package structures. Enter the concept of short-naming, using a well defined and well known "short name" for the impl class. For example, this JdbcTransactionFactory is registered under the short name "jdbc", so:

hibernate.transaction.factory\_class=jdbc

is functionally equivalent to the initial example. Not only is the second form more concise, it is also upgrade proof.

The short name mappings in this service can be managed, even by applications and integrators which can be very powerful. For more information on this aspect, see:

* BootstrapServiceRegistryBuilder#withStrategySelector
* BootstrapServiceRegistryBuilder#withStrategySelectors
* org.hibernate.boot.registry.selector.StrategyRegistrationProvider (via ServiceLoader discovery)
* 'StrategySelector#registerStrategyImplementor` / 'StrategySelector#unRegisterStrategyImplementor`

The service role for this service is org.hibernate.boot.registry.selector.spi.StrategySelector.

StandardServiceRegistry

The org.hibernate.boot.registry.StandardServiceRegistry defines the main Hibernate ServiceRegistry, building on the BootstrapServiceRegistry (BootstrapServiceRegistry is its parent). This registry is generally built using the org.hibernate.boot.registry.StandardServiceRegistryBuilder class. By default it holds most of the Services used by Hibernate. For the full list of Services typically held in the StandardServiceRegistry, see the source code of org.hibernate.service.StandardServiceInitiators. Some particular StandardServiceRegistry Services of note include:

ConnectionProvider/MultiTenantConnectionProvider

The Service providing Hibernate with Connections as needed. Comes in 2 distinct (and mutually exclusive) roles:

* org.hibernate.engine.jdbc.connections.spi.ConnectionProvider provides Connections in normal environments.
* org.hibernate.engine.jdbc.connections.spi.MultiTenantConnectionProvider provides (tenant-specific) Connections in multi-tenant environments.

JdbcServices

org.hibernate.engine.jdbc.spi.JdbcServices is an aggregator Service (a Service that aggregates other Services) exposing unified functionality around JDBC accessibility.

TransactionFactory

org.hibernate.engine.transaction.spi.TransactionFactory is used to tell Hibernate how to control or integrate with transactions.

JtaPlatform

When using a JTA-based TransactionFactory, the org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform Service provides Hibernate access to the JTA TransactionManager and UserTransaction, as well handling Synchronization registration.

Here are the steps (in order of precedence) that Hibernate follows to determine the JtaPlatform to use:

1. Explicit setting keyed as "hibernate.transaction.jta.platform" which can refer to
   * a JtaPlatform instance
   * a Class<? extends JtaPlatform> reference
   * the name (see StrategySelector service) of a JtaPlatform strategy
   * the FQN of a JtaPlatform implementation
2. Discover via the org.hibernate.engine.transaction.jta.platform.spi.JtaPlatformResolver Service, which by default:
   * looks for org.hibernate.engine.transaction.jta.platform.spi.JtaPlatformProvider implementations via ServiceLoader, if one is found its reported JtaPlatform is used (first wins).
   * Attempts a number of well-known Class lookups for various environments.

RegionFactory

This is the second level cache service in terms of starting the underlying cache provider

SessionFactoryServiceRegistryFactory

org.hibernate.service.spi.SessionFactoryServiceRegistryFactory is a service that acts as a factory for building the third type of ServiceRegistry, SessionFactoryServiceRegistry which we will discuss next. I opted for the "factory as service" approach because in the current design there is really not a good exposed hook-in spot for when the SessionFactoryServiceRegistry needs to be built. This may very well change in 5.0

SessionFactoryServiceRegistry

org.hibernate.service.spi.SessionFactoryServiceRegistry is the 3rd standard Hibernate ServiceRegistry. Typically, its parent registry is the StandardServiceRegistry. SessionFactoryServiceRegistry is designed to hold Services which need access to the SessionFactory. Currently that is just 3 Services.

|  |  |
| --- | --- |
|  | Integrators, as it stands in 4.x, operate on the SessionFactoryServiceRegistry… |

EventListenerRegistry

org.hibernate.event.service.spi.EventListenerRegistry is the big Service managed in the SessionFactoryServiceRegistry. This is the Service that manages and exposes all of Hibernate’s event listeners. A major use-case for Integrators is to alter the listener registry.

If doing custom listener registration, it is important to understand the org.hibernate.event.service.spi.DuplicationStrategy and its effect on registration. The basic idea is to tell Hibernate:

* what makes a listener a duplicate
* how to handle duplicate registrations (error, first wins, last wins)

StatisticsImplementor

org.hibernate.stat.spi.StatisticsImplementor is the SPI portion of the org.hibernate.stat.Statistics API. The collector portion, if you will.

Service lifecycle

Managing the lifecycle of services is the big role of a ServiceRegistry as a container for those services. The overall lifecycle of a Service is:

1. [initiation](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-initiation)
2. (optional) [configuration](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-configuration)
3. (optional) [starting](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-starting)
4. in use - until registry closed
5. (optional) [stopping](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-stopping)

Initiation (creation)

A Service needs to be initiated/created. We’ll explore the details a little more when we discuss [Building ServiceRegistry](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#serviceregistry-building). But generally speaking, either

* a Service can be instantiated directly and handed to the ServiceRegistry
* A ServiceInitiator can be handed to the ServiceRegistry to initiate the Service on-demand.

Configuration

A Service can optionally implement the org.hibernate.service.spi.Configurable interface to be handed the java.util.Map of configuration settings handed to Hibernate during initial bootstrapping. Configurable#configure is called after initiation but before usage

Starting

A Service can optionally implement org.hibernate.service.spi.Startable to receive a callback just prior to going into "in use". Reflexively speaking, it is generally good practice for a Service needing Startable to also need Stoppable ([stopping](http://docs.jboss.org/hibernate/orm/4.3/topical/html/registries/ServiceRegistries.html#service-stopping)).

Stopping

A Service can optionally implement org.hibernate.service.spi.Stoppable to receive a callback as the Service is taken out of "in use" as part ServiceRegistry shutdown.

Manageable (JMX)

A Service can optionally implement org.hibernate.service.spi.Manageable to be made available to JMX.

|  |  |
| --- | --- |
|  | This particular feature is still under design/development |

Service Dependencies

Services sometimes depend on other services. For example, the DataSourceConnectionProvider service implementation usually needs access to the JndiService to perform JNDI lookups. This has 2 implications. First, it means that DataSourceConnectionProvider needs access to JndiService. Secondly it means that the JndiService musty be fully "in use" prior to its usage from DataSourceConnectionProvider.

There are 2 ways to obtain access to dependent Services:

1. Have the Service implement org.hibernate.service.spi.ServiceRegistryAwareService, which will inject the ServiceRegistry into your Service. You can then look up any Services you need access to. The returned Services you lookup will be fully ready for use.
2. Injecting specific Services using @org.hibernate.service.spi.InjectService.
   1. The Service role to inject is generally inferred by the type of parameter of the method to which the annotation is attached. If the parameter type is different from the Service role, use InjectService#serviceRole to name the role explicitly.
   2. By default the Service to inject is considered required (an exception will be thrown if it is not found). If the service to be injected is optional, use InjectService#required=false.

Building ServiceRegistry

Once built, a ServiceRegistry is generally considered immutable. The Services themselves might accept re-configuration, but immutability here means adding/replacing services. So all the services hosted in a particular ServiceRegistry must be known up-front. To this end, building a ServiceRegistry usually employees a [builder](http://en.wikipedia.org/wiki/Builder_pattern).

Building BootstrapServiceRegistry

Building the BootstrapServiceRegistry is normally done via the 'org.hibernate.boot.registry.BootstrapServiceRegistryBuilder` class which exposes methods for defining ClassLoaders to use, non-discoverable Integrators to incorporate, etc.

By default Hibernate will use the Thread-context ClassLoader (TCCL), if one, as well as the ClassLoader of its classes as the ClassLoaders it will consult when asked to load classes or resources or to perform ServiceLoader resolutions. You can tell Hibernate to consider any additional ClassLoaders via the overloaded BootstrapServiceRegistryBuilder#with(ClassLoader) method:

BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()

.with( anAdditionalClassLoader )

.with( anotherAdditionalClassLoader )

.build();

|  |  |
| --- | --- |
|  | you can also tell Hibernate to use a completely different ClassLoaderService implementation usingBootstrapServiceRegistryBuilder#with(ClassLoaderService). |

Integrators are normally discovered via the JDK ServiceLoader mechanism. To tell Hibernate about an Integrator that will not be discovered (for whatever reason) you would use the BootstrapServiceRegistryBuilder#with(Integrator) method:

BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()

.with( new MyCustomIntegrator() )

.build();

BootstrapServiceRegistryBuilder also exposes methods to add extra strategy selections. Let’s say we developed a custom CORBA-based TransactionFactory named CORBATransactionFactory and that we’d like to make this available via short-naming. One option would be to explicitly set up the short name during BootstrapServiceRegistry building:

BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()

.withStrategySelector( TransactionFactory.class, "corba", CORBATransactionFactory.class )

.build();

If we were going to distribute our CORBATransactionFactory, we might develop aorg.hibernate.boot.registry.selector.StrategyRegistrationProvider:

public class CORBATransactionFactoryStrategyRegistrationProvider implements StrategyRegistrationProvider {

public Iterable<StrategyRegistration> getStrategyRegistrations() {

return Collections.singletonList(

(StrategyRegistration) new SimpleStrategyRegistrationImpl<TransactionFactory>(

TransactionFactory.class,

CORBATransactionFactory.class,

"corba"

)

);

}

}

which we could register explicitly:

BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()

.withStrategySelectors( new CORBATransactionFactoryStrategyRegistrationProvider() )

.build();

or define for discovery by adding a META-INF/services/org.hibernate.boot.registry.selector.StrategyRegistrationProvider file to our artifact naming CORBATransactionFactoryStrategyRegistrationProvider.

We might combine several of these at once:

BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()

.with( anAdditionalClassLoader )

.with( anotherAdditionalClassLoader )

.with( new MyCustomIntegrator() )

.withStrategySelector( ConnectionProvider.class, "custom", MyCustomConnectionProvider.class )

.withStrategySelectors( new CORBATransactionFactoryStrategyRegistrationProvider() )

.build();

Building StandardServiceRegistry

Building the StandardServiceRegistry is normally done via the 'org.hibernate.boot.registry.StandardServiceRegistryBuilder` which exposes methods for managing settings and controlling the services hosted by the built StandardServiceRegistry.

Managing settings can be as simple as telling the builder about one or more settings directly:

StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.applySetting( "hibernate.hbm2ddl.auto", true )

.applySettings( Collections.singletonMap( "hibernate.transaction.factory\_class", "jdbc" ) )

.build();

Or we can tell it to load settings from various files:

StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.configure()

.configure( "com/acme/hibernate.cfg.xml" )

.loadProperties( "com/acme/hibernate.properties" )

.build();

|  |  |
| --- | --- |
|  | loads settings from an XML file (conforming to the Hibernate cfg.xml DTD) via a ClassLoader resource lookup for hibernate.cfg.xml |
|  | loads settings from an XML file (conforming to the Hibernate cfg.xml DTD) via a ClassLoader resource lookup for com/acme/hibernate.cfg.xml |
|  | loads settings from Properties via a ClassLoader resource lookup for com/acme/hibernate.properties |

The other methods of interest on StandardServiceRegistryBuilder relate to customizing the Services to use. We can either pass in a Service instance to use or the ServiceInitiator to use as already discussed. There are 2 distinct ways to customize the Services to use:

Building StandardServiceRegistry - Overriding

Here the intent is to override or replace a service impl. Many of the standard ServiceInitiators look through the settings to determine the appropriate service to use. Going back to an example we have used multiple times:

hibernate.transaction.factory\_class=jdbc

The standard TransactionFactoryInitiator looks for this setting and determines what TransactionFactory service implementation to use. Let’s say for whatever reason we always want it to use JdbcTransactionFactory:

StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.addService( TransactionFactory.class, new JdbcTransactionFactory() )

.build();

Or say we want to resolve the service implementation to use differently:

StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.addInitiator( new MyCustomTransactionFactoryInitiator() )

.build();

Building StandardServiceRegistry - Expanding

Here the intent is to have the ServiceRegistry host custom services (completely new Service roles). As an example, let’s say our application publishes Hibernate events to a JMS Topic and that we want to leverage the Hibernate ServiceRegistry to host a Service representing our TopicPublisher. So we will expand the ServiceRegistry to host this completely new Service role:

/\*\*

\* The service role

\*/

public interface EventPublishingService extends Service {

public void publish(Event theEvent);

}

/\*\*

\* A disabled (no-op) impl

\*/

public class DisabledEventPublishingServiceImpl implements EventPublishingService {

public static DisabledEventPublishingServiceImpl INSTANCE = new DisabledEventPublishingServiceImpl();

private DisabledEventPublishingServiceImpl() {

}

@Override

public void publish(Event theEvent) {

// nothing to do...

}

}

/\*\*

\* A standard impl

\*/

public class EventPublishingServiceImpl

implements EventPublishingService, Configurable, Startable, Stoppable, ServiceRegistryAwareService {

private ServiceRegistryImplementor serviceRegistry;

private String jmsConnectionFactoryName;

private String destinationName;

private Connection jmsConnection;

private Session jmsSession;

private MessageProducer publisher;

@Override

public void injectServices(ServiceRegistryImplementor serviceRegistry) {

this.serviceRegistry = serviceRegistry;

}

public void configure(Map configurationValues) {

this.jmsConnectionFactoryName = configurationValues.get( JMS\_CONNECTION\_FACTORY\_NAME\_SETTING );

this.destinationName = configurationValues.get( JMS\_DESTINATION\_NAME\_SETTING );

}

@Override

public void start() {

final JndiService jndiService = serviceRegistry.getService( JndiService.class );

final ConnectionFactory jmsConnectionFactory = jndiService.locate( jmsConnectionFactoryName );

this.jmsConnection = jmsConnectionFactory.createConnection();

this.jmsSession = jmsConnection.createSession( true, Session.AUTO\_ACKNOWLEDGE );

final Destination destination = jndiService.locate( destinationName );

this.publisher = jmsSession.createProducer( destination );

}

@Override

public void publish(Event theEvent) {

publisher.send( theEvent );

}

@Override

public void stop() {

publisher.close();

jmsSession.close();

jmsConnection.close();

}

}

public class EventPublishingServiceInitiator implements StandardServiceInitiator<EventPublishingService> {

public static EventPublishingServiceInitiator INSTANCE = new EventPublishingServiceInitiator();

public static final String ENABLE\_PUBLISHING\_SETTING = "com.acme.EventPublishingService.enabled";

public Class<R> getServiceInitiated() {

return EventPublishingService.class;

}

@Override

public R initiateService(Map configurationValues, ServiceRegistryImplementor registry) {

final boolean enabled = extractBoolean( configurationValues, ENABLE\_PUBLISHING\_SETTING );

if ( enabled ) {

return new EventPublishingServiceImpl();

}

else {

return DisabledEventPublishingServiceImpl.INSTANCE;

}

}

}

Now, lets tell Hibernate about this custom Service:

StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()

.addInitiator( EventPublishingServiceInitiator.INSTANCE )

...

.build();

Hibernate openSession VS openStatelessSession

We mostly use hibernate ‘s *sessionFactory.openSession()* method which keeps track of all events of hibernate along with managing first level cache, and also interacts with second level cache of hibernate. It also performs operations cascade on associated entities.

In *openSession*() method dirty checking is done automatically,  means when we fetch data from DB and do some changes it automatically check updated record and update the database accordingly.Many of things are not possible to do with *org.hibernate.Session* interface.

However, Hibernate gives alternative interface when you want to work with your database like executing simple JDBC Statements. This interface is *org.hibernate.StatelessSession*.

Hibernate *sessionFactory.openStatelessSession()*method works line simple JDBC. It neither listens any of hibernate events nor does any things by own.

A StatelessSession does not maintain first level cache or never interacts with second level cache, nor does it automatically do dirty checking, nor do any operation cascade on related associations. In StatelessSession you have to explicitly trigger a query or event for each update or modification on an Object.

Hibernate SessionFactory provides three methods through which we can get Session object – getCurrentSession(), openSession() and openStatelessSession().

**Hibernate SessionFactory getCurrentSession**

Hibernate SessionFactory getCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file like below.

<property name="hibernate.current\_session\_context\_class">thread</property>

If its not configured to thread, then we will get below exception.

Exception in thread "main" org.hibernate.HibernateException: No CurrentSessionContext configured!

at org.hibernate.internal.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:1012)

at com.journaldev.hibernate.main.HibernateSessionExample.main(HibernateSessionExample.java:16)

Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

Hibernate Session objects are not thread safe, so we should not use it in multi-threaded environment. We can use it in single threaded environment because it’s relatively faster than opening a new session.

**Hibernate SessionFactory openSession**

Hibernate SessionFactory openSession() method always opens a new session. We should close this session object once we are done with all the database operations.

We should open a new session for each request in multi-threaded environment. For web application frameworks, we can choose to open a new session for each request or for each session based on the requirement.

**Hibernate SessionFactory openStatelessSession**

Hibernate SessionFactory openStatelessSession() method returns instance of StatelessSession. There is another overloaded method where we can pass java.sql.Connection object to get a stateless session object from hibernate.

StatelessSession in Hibernate does not implement first-level cache and it doesn’t interact with any second-level cache. Since it’s stateless, it doesn’t implement transactional write-behind or automatic dirty checking or do cascading operations to associated entities.

**Collections** are also **ignored** by a stateless session. Operations performed via a stateless session bypass Hibernate’s event model and interceptors. It’s more like a normal JDBC connection and doesn’t provide any benefits that come from using hibernate framework.

However, stateless session can be a good fit in certain situations. For example where we are loading bulk data into database and we don’t want hibernate session to hold huge data in first-level cache memory.

**What does Session lock() method do in Hibernate?**

This one is one of the tricky Hibernate Interview questions because Session's lock() method reattach object without synchronizing or updating with the database. So you need to be very careful while using lock() method. By the way, you can always use Session's update() method to sync with the database during reattachment. Sometimes this Hibernate question is also asked as *what is difference between Session's lock() and update() method*. You can use this key point to answer that question as well

# 

# [Difference Between session.evict vs clear](http://stackoverflow.com/questions/10476488/difference-between-session-evict-vs-clear)

|  |  |
| --- | --- |
|  | [evict()](http://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/Session.html#evict%28java.lang.Object%29) evicts a single object from the session. [clear()](http://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/Session.html#clear%28%29) evicts all the objects in the session. Calling clear() is like calling evict() on every object associated with the session. |

* If a child is dereferenced by a persistent parent, nothing special happens - the application should explicitly delete the child if necessary - unless cascade="**delete-orphan**", in which case the "orphaned" child is deleted.

So, the long story short here is:

FetchType.LAZY = Doesn’t load the relationships unless explicitly “asked for” via getter  
FetchType.EAGER = Loads ALL relationships

1. @OneToOne(fetch=FetchType.EAGER)

One way to load :

using Hibernate.initialize(). This is often more straightforward than it sounds.

## Dynamic association fetching

You can specify association fetching semantics at runtime using setFetchMode().

List cats = sess.createCriteria(Cat.**class**)

    .add( Restrictions.like("name", "Fritz%") )

    .setFetchMode("mate", FetchMode.EAGER)

    .setFetchMode("kittens", FetchMode.EAGER)

    .list();

This query will fetch both mate and kittens by outer join. See [Section 21.1, “Fetching strategies”](https://docs.jboss.org/hibernate/orm/4.0/manual/en-US/html/performance.html#performance-fetching) for more information.

**Detached queries and subqueries**

The DetachedCriteria class allows you to create a query outside the scope of a session and then execute it using an arbitrary Session.

DetachedCriteria query = DetachedCriteria.forClass(Cat.**class**)

    .add( Property.forName("sex").eq('F') );

Session session = ....;

Transaction txn = session.beginTransaction();

List results = query.getExecutableCriteria(session).setMaxResults(100).list();

txn.commit();

session.close();

A DetachedCriteria can also be used to express a subquery. Criterion instances involving subqueries can be obtained via Subqueries or Property.

DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.**class**)

    .setProjection( Property.forName("weight").avg() );

session.createCriteria(Cat.**class**)

    .add( Property.forName("weight").gt(avgWeight) )

    .list();

DetachedCriteria weights = DetachedCriteria.forClass(Cat.**class**)

    .setProjection( Property.forName("weight") );

session.createCriteria(Cat.**class**)

    .add( Subqueries.geAll("weight", weights) )

    .list();

Correlated subqueries are also possible:

DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.**class**, "cat2")

    .setProjection( Property.forName("weight").avg() )

    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );

session.createCriteria(Cat.**class**, "cat")

    .add( Property.forName("weight").gt(avgWeightForSex) )

    .list();

Example of multi-column restriction based on a subquery:

DetachedCriteria sizeQuery = DetachedCriteria.forClass( Man.**class** )

    .setProjection( Projections.projectionList().add( Projections.property( "weight" ) )

                                                .add( Projections.property( "height" ) ) )

    .add( Restrictions.eq( "name", "John" ) );

session.createCriteria( Woman.**class** )

    .add( Subqueries.propertiesEq( **new** String[] { "weight", "height" }, sizeQuery ) )

    .list();

**14.1. Interceptors**

The Interceptor interface provides callbacks from the session to the application, allowing the application to inspect and/or manipulate properties of a persistent object before it is saved, updated, deleted or loaded. One possible use for this is to track auditing information. For example, the following Interceptor automatically sets the createTimestamp when an Auditable is created and updates the lastUpdateTimestamp property when an Auditable is updated.

You can either implement Interceptor directly or extend EmptyInterceptor.

**package** org.hibernate.test;

**import** java.io.Serializable;

**import** java.util.Date;

**import** java.util.Iterator;

**import** org.hibernate.EmptyInterceptor;

**import** org.hibernate.Transaction;

**import** org.hibernate.type.Type;

**public** **class** AuditInterceptor **extends** EmptyInterceptor {

**private** int updates;

**private** int creates;

**private** int loads;

**public** void onDelete(Object entity,

                         Serializable id,

                         Object[] state,

                         String[] propertyNames,

                         Type[] types) {

        // **do** nothing

    }

**public** boolean onFlushDirty(Object entity,

                                Serializable id,

                                Object[] currentState,

                                Object[] previousState,

                                String[] propertyNames,

                                Type[] types) {

**if** ( entity **instanceof** Auditable ) {

            updates++;

**for** ( int i=0; i < propertyNames.length; i++ ) {

**if** ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {

                    currentState[i] = **new** Date();

**return** true;

                }

            }

        }

**return** false;

    }

**public** boolean onLoad(Object entity,

                          Serializable id,

                          Object[] state,

                          String[] propertyNames,

                          Type[] types) {

**if** ( entity **instanceof** Auditable ) {

            loads++;

        }

**return** false;

    }

**public** boolean onSave(Object entity,

                          Serializable id,

                          Object[] state,

                          String[] propertyNames,

                          Type[] types) {

**if** ( entity **instanceof** Auditable ) {

            creates++;

**for** ( int i=0; i<propertyNames.length; i++ ) {

**if** ( "createTimestamp".equals( propertyNames[i] ) ) {

                    state[i] = **new** Date();

**return** true;

                }

            }

        }

**return** false;

    }

**public** void afterTransactionCompletion(Transaction tx) {

**if** ( tx.wasCommitted() ) {

            System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);

        }

        updates=0;

        creates=0;

        loads=0;

    }

}

There are two kinds of inteceptors: Session-scoped and SessionFactory-scoped.

A Session-scoped interceptor is specified when a session is opened using one of the overloaded SessionFactory.openSession() methods accepting an Interceptor.

Session session = sf.openSession( **new** AuditInterceptor() );

A SessionFactory-scoped interceptor is registered with the Configuration object prior to building the SessionFactory. Unless a session is opened explicitly specifying the interceptor to use, the supplied interceptor will be applied to all sessions opened from that SessionFactory. SessionFactory-scoped interceptors must be thread safe. Ensure that you do not store session-specific states, since multiple sessions will use this interceptor potentially concurrently.

**new** Configuration().setInterceptor( **new** AuditInterceptor() );

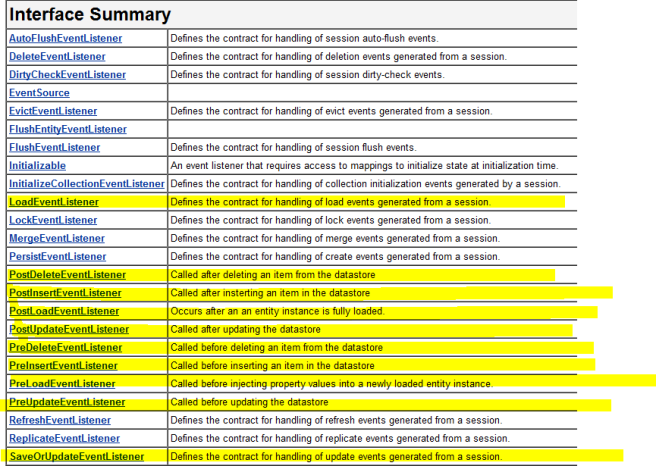
**Event system**

If you have to react to particular events in your persistence layer, you can also use the Hibernate *event*architecture. The event system can be used in addition, or as a replacement, for interceptors.

Many methods of the Session interface correlate to an event type. The full range of defined event types is declared as enum values on org.hibernate.event.spi.EventType. When a request is made of one of these methods, the Hibernate Session generates an appropriate event and passes it to the configured event listeners for that type. Out-of-the-box, these listeners implement the same processing in which those methods always resulted. However, you are free to implement a customization of one of the listener interfaces (i.e., the LoadEvent is processed by the registered implementation of the LoadEventListener interface), in which case their implementation would be responsible for processing any load() requests made of the Session.

Below is the commonly used listener interfaces in hibernate.

Event Listeners in org.hibernate.event package



Along with these listeners, you can use associated event objects to get information related with a particular event. For instance, after updating an object you can retrieve the updated instance of an object using PostUpdateEvent object.  Then it can be used to set the properties like update timestamp.

Below is the commonly used event objects associated with event listeners.

**How to use hibernate events in your application?**

1. Implement the appropriate event interface in your application.
2. Register the listeners in hibernate.cfg.xml file.

Very simple application to demonstrate the post-update event type

Here I’m going to create an application to trap an object after updating.

package com.dunithd.events;

import org.hibernate.event.PostUpdateEvent;

import org.hibernate.event.PostUpdateEventListener;

import com.dunithd.events.InventoryProduct;

/\*\*

\* A class that implements PostUpdateEventListener interface

\*

\* @author duischen

\* @version 1.0, Oct 27, 2009

\*/

public class ProductUpdateListener implements PostUpdateEventListener {

@Override

public void onPostUpdate(PostUpdateEvent event) {

if (event.getEntity() instanceof InventoryProduct) {

InventoryProduct product = (InventoryProduct) event.getEntity();

// Check for quantity and re-order level for this product

int quantity = product.getQuantity();

int reOrderLevel = product.getReOrderLevel();

// Check if this product is under the re-order level

if (quantity <= reOrderLevel) {

System.out.println("Needurchase.");

} else {

System.out.println("No to purchase.");

}

}

}

}

Then register this listener in hibernate.cfg.xml file.

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

.....

<mapping resource="com/dunithd/events/InventoryProduct.hbm.xml" />

<!-- Register the listeners -->

<listener type="post-update"/>

</session-factory>

</hibernate-configuration>

Create the database table as follows.

create table products (

int product\_id primary key,

varchar(100) description,

float price,

int quantity,

int reorder\_level

)

When you run your application, you can see the message “Need to purchase.” or “No need to purchase”, depending on the quantity of the product.

Hibernate 5.0:

*Example 194. Native Bootstrapping - Putting it all together*

StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()

.configure( "org/hibernate/example/hibernate.cfg.xml" )

.build();

Metadata metadata = new MetadataSources( standardRegistry )

.addAnnotatedClass( MyEntity.class )

.addAnnotatedClassName( "org.hibernate.example.Customer" )

.addResource( "org/hibernate/example/Order.hbm.xml" )

.addResource( "org/hibernate/example/Product.orm.xml" )

.getMetadataBuilder()

.applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )

.build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()

.applyBeanManager( getBeanManager() )

.build();

#### Building the SessionFactory

The final step in native bootstrapping is to build the SessionFactory itself. Much like discussed above, if you are ok with the default behavior of building a SessionFactory from a Metadata reference, you can simply call the [buildSessionFactory](https://docs.jboss.org/hibernate/orm/5.0/javadocs/org/hibernate/boot/Metadata.html" \l "buildSessionFactory--) method on the Metadata object.

However, if you would like to adjust that building process you will need to use SessionFactoryBuilder as obtained via [Metadata#getSessionFactoryBuilder. Again, see its [Javadocs](https://docs.jboss.org/hibernate/orm/5.0/javadocs/org/hibernate/boot/Metadata.html" \l "getSessionFactoryBuilder--) for more details.

*Example 194. Native Bootstrapping - Putting it all together*

StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()

.configure( "org/hibernate/example/hibernate.cfg.xml" )

.build();

Metadata metadata = new MetadataSources( standardRegistry )

.addAnnotatedClass( MyEntity.class )

.addAnnotatedClassName( "org.hibernate.example.Customer" )

.addResource( "org/hibernate/example/Order.hbm.xml" )

.addResource( "org/hibernate/example/Product.orm.xml" )

.getMetadataBuilder()

.applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )

.build();

SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()

.applyBeanManager( getBeanManager() )

.build();

The bootstrapping API is quite flexible, but in most cases it makes the most sense to think of it as a 3 step process:

1. Build the StandardServiceRegistry
2. Build the Metadata
3. Use those 2 to build the SessionFactory

*Example 195. Building SessionFactory via SessionFactoryBuilder*

StandardServiceRegistry standardRegistry = new StandardServiceRegistryBuilder()

.configure( "org/hibernate/example/hibernate.cfg.xml" )

.build();

Metadata metadata = new MetadataSources( standardRegistry )

.addAnnotatedClass( MyEntity.class )

.addAnnotatedClassName( "org.hibernate.example.Customer" )

.addResource( "org/hibernate/example/Order.hbm.xml" )

.addResource( "org/hibernate/example/Product.orm.xml" )

.getMetadataBuilder()

.applyImplicitNamingStrategy( ImplicitNamingStrategyJpaCompliantImpl.INSTANCE )

.build();

SessionFactoryBuilder sessionFactoryBuilder = metadata.getSessionFactoryBuilder();

// Supply an SessionFactory-level Interceptor

sessionFactoryBuilder.applyInterceptor( new CustomSessionFactoryInterceptor() );

// Add a custom observer

sessionFactoryBuilder.addSessionFactoryObservers( new CustomSessionFactoryObserver() );

// Apply a CDI BeanManager ( for JPA event listeners )

sessionFactoryBuilder.applyBeanManager( getBeanManager() );

SessionFactory sessionFactory = sessionFactoryBuilder.build();

### Accessing Hibernate APIs from JPA

JPA defines an incredibly useful method to allow applications access to the APIs of the underlying provider.

*Example 201. Accessing Hibernate APIs from JPA*

Session session = entityManager.unwrap( Session.class );

SessionImplementor sessionImplementor = entityManager.unwrap( SessionImplementor.class );

SessionFactory sessionFactory = entityManager.getEntityManagerFactory().unwrap( SessionFactory.class );