Services

Services provide various types of functionalities, in a pluggable manner. Specifically, they are interfaces defining certain functionality and then implementations of those service contract interface. 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.

Note: - 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.

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

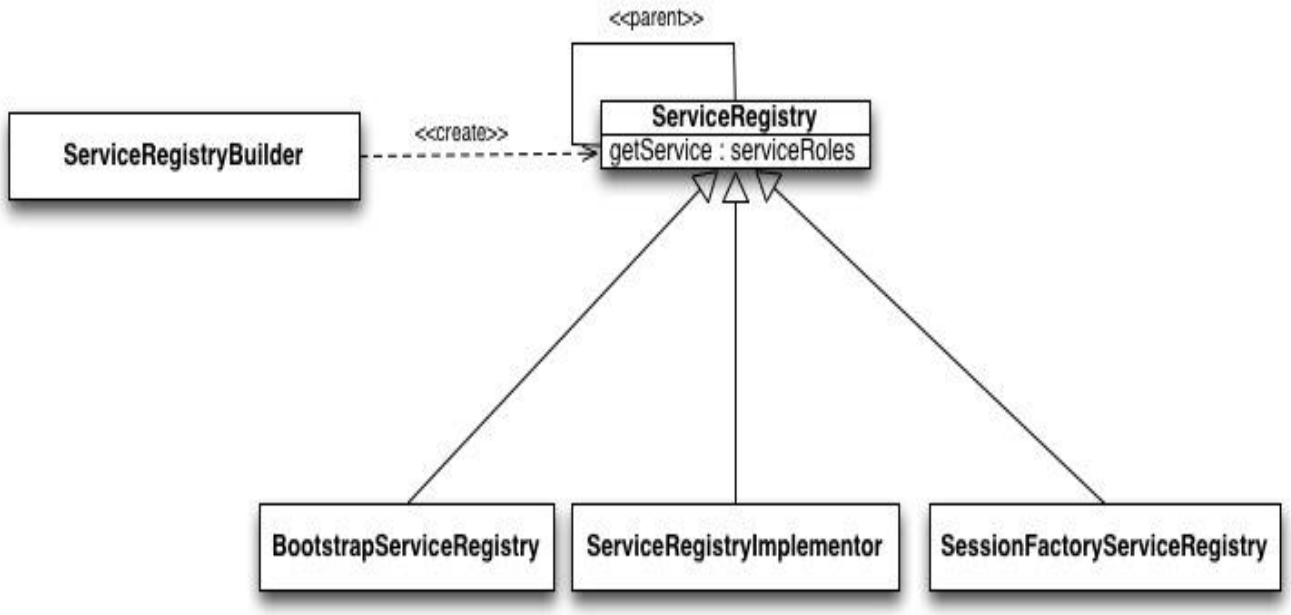
Internally Hibernate always references org.hibernate.engine.jdbc.connections.spi.ConnectionProvider rather than specific implementations in consuming the service. Because of that fact, other ConnectionProvider service implementations could be plugged in.

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.

What is a ServiceRegistry?

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

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 the **org.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

- **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.AbstractServiceRegistryImpl** convenience 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**
- **StandardServiceRegistry**
- **SessionFactoryServiceRegistry**

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

- **ClassLoaderService: -** allows Hibernate to interact with the *ClassLoader* of the various runtime environments
- **IntegratorService: -** controls the discovery and management of the *Integrator* service allowing third-party applications to integrate with Hibernate
- **StrategySelector: -** resolves implementations of various strategy contracts

To build a *BootstrapServiceRegistry* implementation, we use the *BootstrapServiceRegistryBuilder* factory class, which allows customizing these three services in a type-safe manner: -

```
BootstrapServiceRegistry bootstrapServiceRegistry = new BootstrapServiceRegistryBuilder()
        .applyClassLoader()
        .applyIntegrator()
        .applyStrategySelector()
        .build();
```

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.

Like the previous registry, we use the ***StandardServiceRegistryBuilder*** to create an instance of the ***StandardServiceRegistry****:*

```
StandardServiceRegistryBuilder standardServiceRegistry = new
StandardServiceRegistryBuilder();
```

Under the hood, the *StandardServiceRegistryBuilder* creates and uses an instance of *BootstrapServiceRegistry.* We can also use an overloaded constructor to pass an already created instance:

```
BootstrapServiceRegistry bootstrapServiceRegistry = new
BootstrapServiceRegistryBuilder().build();

StandardServiceRegistryBuilder standardServiceRegistryBuilder = new
StandardServiceRegistryBuilder(bootstrapServiceRegistry);
```

We use this builder to load a configuration from a resource file, such as the default *hibernate.cfg.xml*, and finally, we invoke the *build()* method to get an instance of the *StandardServiceRegistry.*

```
StandardServiceRegistry standardServiceRegistry =
standardServiceRegistryBuilder .configure() .build();
```

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.

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 Hibernates 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)

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:

- initiation
- configuration (optional)
- starting (optional)
- in use - until registry closed
- stopping (optional)

Initiation (creation)

A Service needs to be initiated/created.

- 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).

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.

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:

- 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.
- Injecting specific Services using **@org.hibernate.service.spi.InjectService.**

a. 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.
b. 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.**

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.

There are 2 ways an integrator becomes known.

- The integrator may be manually registered by calling **BootstrapServiceRegistryBuilder#with(Integrator)**
- 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. 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 the **org.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
```

ConnectionProvider/MultiTenantConnectionProvider

The Service providing Hibernate with Connections as needed.

It 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.**