Introduction to ORM with Spring

The Spring Framework supports integration with the Java Persistence API (JPA) and supports native Hibernate for resource management, data access object (DAO) implementations, and transaction strategies. For example, for Hibernate, there is first-class support with several convenient IoC features that address many typical Hibernate integration issues. You can configure all of the supported features for OR (object relational) mapping tools through Dependency Injection. They can participate in Spring's resource and transaction management, and they comply with Spring's generic transaction and DAO exception hierarchies. The recommended integration style is to code DAOs against plain Hibernate or JPA APIs.

Spring adds significant enhancements to the ORM layer of your choice when you create data access applications. You can leverage as much of the integration support as you wish, and you should compare this integration effort with the cost and risk of building a similar infrastructure in-house. You can use much of the ORM support as you would a library, regardless of technology, because everything is designed as a set of reusable JavaBeans. ORM in a Spring IoC container facilitates configuration and deployment. Thus, most examples in this section show configuration inside a Spring container.

The benefits of using the Spring Framework to create your ORM DAOs include:

- **Easier testing.** Spring's IoC approach makes it easy to swap the implementations and configuration locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, transaction managers, and mapped object implementations (if needed). This in turn makes it much easier to test each piece of persistence-related code in isolation.
- **Common data access exceptions.** Spring can wrap exceptions from your ORM tool, converting them from proprietary (potentially checked) exceptions to a common runtime `DataAccessException` hierarchy. This feature lets you handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches, throws, and exception declarations. You can still trap

Spring Framework
6.2.7
Search CTRL + k
Overview
Core Technologies
Data Access
Transaction Management
DAO Support
Data Access with JDBC
Data Access with R2DBC
Object Relational Mapping (ORM) Data Access
Introduction to ORM with Spring
General ORM Integration Considerations
Hibernate
JPA
Marshalling XML by Using Object-XML Mappers
Appendix
Web on Servlet Stack
Web on Reactive Stack
Testing
Integration
Language Support
Appendix
Java API
Kotlin API

Edit this Page
GitHub Project
Stack Overflow

- **General resource management.** Spring application contexts can handle the location and configuration of Hibernate `SessionFactory` instances, JPA `EntityManagerFactory` instances, JDBC `DataSource` instances, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy, and safe handling of persistence resources. For example, related code that uses Hibernate generally needs to use the same Hibernate `Session` to ensure efficiency and proper transaction handling. Spring makes it easy to create and bind a `Session` to the current thread transparently, by exposing a current `Session` through the Hibernate `SessionFactory`. Thus, Spring solves many chronic problems of typical Hibernate usage, for any local or JTA transaction environment.
- **Integrated transaction management.** You can wrap your ORM code with a declarative, aspect-oriented programming (AOP) style method interceptor either through the `@Transactional` annotation or by explicitly configuring the transaction AOP advice in an XML configuration file. In both cases, transaction semantics and exception handling (rollback and so on) are handled for you. As discussed in Resource and Transaction Management, you can also swap various transaction managers, without affecting your ORM-related code. For example, you can swap between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. Additionally, JDBC-related code can fully integrate transactionally with the code you use to do ORM. This is useful for data access that is not suitable for ORM (such as batch processing and BLOB streaming) but that still needs to share common transactions with ORM operations.

TIP

For more comprehensive ORM support, including support for alternative database technologies such as MongoDB, you might want to check out the Spring Data suite of projects. If you are a JPA user, the Getting Started Accessing Data with JPA guide from spring.io provides a great introduction.