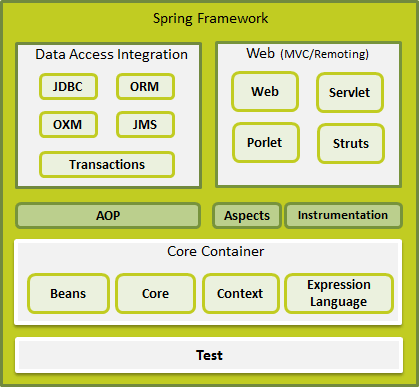
**Springs**

Spring is framework focussed on enterprise java development through dependency injection, aspect-oriented programming and boiler plate reduction.

Spring framework composed of several modules.

**Spring framework architecture diagram**



**Spring Core Container:**

It manages the how the beans in spring based application are created, configured and managed.

**Spring AOP:**

Spring supports aspect-oriented programming in its AOP module.

**Data access and integration:**

This module responsible for accessing the data either through JDBC or ORM (hibernate)

**Spring container**

* It is core of spring framework
* It uses dependency injection to manage components.
* It creates association between components.



Spring comes with several container implementations that can be categorized into two distinct types

**Bean factories** provide basic support for DI.

**Application contexts** which provides application framework services

Application contexts had more advantages than Bean factories.

**Application contexts**:

ClassPathXmlApplicationContext: Loads a context definition from XML file located in class path.

FileSystemXmlApplicationContext: Loads a context definition from XML file located in file system.

XmlWebApplicationContext: Loads context definitions from XML file contained within Web application.

**Bean life cycle**

1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean’s properties.
3. If the bean implements BeanNameAware, Spring passes the bean’s ID to the set- BeanName() method.
4. If the bean implements BeanFactoryAware, Spring calls the setBeanFactory() method, passing in the bean factory itself. DisposableBean’s destroy() Call custom destroy-method

A bean goes through several steps between creation and destruction in the Spring container. Each step is an opportunity to customize how the bean is managed in Spring.

1. If the bean implements ApplicationContextAware, Spring will call the set- ApplicationContext() method, passing in a reference to the enclosing appli- cation context.
2. If any of the beans implement the BeanPostProcessor interface, Spring calls their postProcessBeforeInitialization() method.
3. If any beans implement the InitializingBean interface, Spring calls their afterPropertiesSet() method. Similarly, if the bean was declared with an init-method, then the specified initialization method will be called.
4. If there are any beans that implement BeanPostProcessor, Spring will call their postProcessAfterInitialization() method.
5. At this point, the bean is ready to be used by the application and will remain in the application context until the application context is destroyed.
6. If any beans implement the DisposableBean interface, then Spring will call their destroy() methods. Likewise, if any bean was declared with a destroy- method, then the specified method will be called.

**Sample XML Configuration**

**Bean Annotations**

@Component: Generic annotation

@Repository: Annotation for repository classes

@Service: Annotation for service classes

@Autowired: Automatically wires the properties

@Qualifier: It helps @Autowired annotation to figure out which bean you want.

Custom annotation: These are alternative to Qualifier annotation.

@Inject: It is JSR specific annotation. Alternative to Autowired

@Named: It is JSR Specific annotation. Alternative to Qualifier

@Value: Annotation for properties values.

|  |  |
| --- | --- |
| Namespace | Purpose |
| aop | Provides elements for declaring aspects and for automatically proxying @AspectJ-annotated classes as Spring aspects |
| beans | The core primitive Spring namespace, enabling declaration of beans and how they should be wired |
| context | Comes with elements for configuring the Spring application context, including the abil- ity to autodetect and autowire beans and injection of objects not directly managed by Spring. |
| jee | Provides configuration elements for declaring message-driven POJOs |
| jms | Provides configuration elements for declaring message-driven POJOs |
| lang | Enables declaration of beans that are implemented as Groovy, JRuby, or BeanShell scripts. |
| mvc | Enables Spring MVC capabilities such as annotation-oriented controllers, view control- lers, and interceptors. |
| oxm | Supports configuration of Spring’s object-to-XML mapping facilities. |
| tx | Provides for declarative transaction configuration. |
| util | A miscellaneous selection of utility elements. Includes the ability to declare collec- tions as beans and support for property placeholder elements. |

**Spring and Java Persistence API**

JPA is emerged out of EJB2 as next generation java persistence standard. It is POJO based persistence mechanism that draws ideas from both hibernate and Java Data Objects (JDO).

The first step towards using JPA with spring is to configure entity manager factory as bean in spring application context file.

**Configuring entity manager factory**

JPA based applications use an implementation of entity manager factory to create instances of entity manger. The JPA Specification defines two kinds of entity managers:

Application-managed: Entity managers are created when an application directly requests one from an entity manager factory. With application-managed entity managers, the application responsible for opening, closing entity managers and involving entity manger transactions. It is most appropriate for standalone applications.

Container-managed: Entity managers are created and managed by Java EE container. Application does not interact with entity manager factory at all. Entity managers are obtained directly through injection or from JNDI. It is appropriate for Java EE.

Application-managed EntityManagers are created by an Entity- ManagerFactory obtained by calling the createEntityManagerFactory() method of the PersistenceProvider. Meanwhile, container-managed EntityManagerFactorys are obtained through PersistenceProvider’s createContainerEntityManager- Factory() method.

* LocalEntityManagerFactoryBean produces an application-managed Entity- ManagerFactory.
* LocalContainerEntityManagerFactoryBean produces a container-managed EntityManagerFactory

**Configuring application managed JPA**

Application managed entity manager factories derive most of their configuration information from configuration file called persistence.xml. This file must appear in META-INF directory with in class path.

Sample bean configuration in spring application context xml file:

CoC

<bean id “entityManagerFactory” class = “org.springframework.orm.jpa.LocalEntityManagerFactoryBean”>

<property name = “persistenceUnitName” value = “<PersistenceUnitName>”/>

</bean>

**Configuring container managed JPA**

When running within a container, an EntityManagerFactory can be produced using information provided by the container—spring, in our case.

Instead of configuring data source details in persistence.xml, you can configure this information in the spring application context. For example, the following <bean> declaration shows how to configure container-managed JPA in spring using LocalContainerEntityManagerFactoryBean

<bean id="emf" class= "org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"> <property name="dataSource" ref="dataSource" /> <property name="jpaVendorAdapter" ref="jpaVendorAdapter" /> </bean>

Here we’ve configured the dataSource property with a Spring-configured data source. Any implementation of javax.sql.DataSource is appropriate, such as those that we configured in section 5.2. Although a data source may still be configured in persistence.xml, the data source specified through this property takes precedence. The jpaVendorAdapter property can be used to provide specifics about the partic- ular JPA implementation to use. Spring comes with a handful of JPA vendor adaptors to choose from:

* EclipseLinkJpaVendorAdapter
* HibernateJpaVendorAdapter
* OpenJpaVendorAdapter
* TopLinkJpaVendorAdapter

In this case, we’re using Hibernate as a JPA implementation, so we’ve configured it with a HibernateJpaVendorAdapter:

<bean id="jpaVendorAdapter" class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="database" value="HSQL" />

<property name="showSql" value="true"/> <property name="generateDdl" value="false"/>

<property name="databasePlatform" value="org.hibernate.dialect.HSQLDialect" />

</bean>

**Spring transaction management support**

Spring like EJB, provides support for both programmatic and declarative transaction management. But spring provides more capabilities than EJB.

Spring support programmatic transactions different from EJB. Spring does not need JTA transaction for single resource application. Spring can support distributed transactions using third party JTA implementations.

Both EJB and spring support declarative transaction demarcation, But spring provides more additional features.

**Choosing transaction manager**

Spring does not directly support transaction manager. It comes with selection transaction managers that delegates responsibility of transaction management.

|  |  |
| --- | --- |
| **Transaction manager** | **Use it when** |
| jca.cci.connection. CciLocalTransactionManager | Using Spring’s support for Java EE Connector Architec- ture (JCA) and the Common Client Interface (CCI). |
| jdbc.datasource. DataSourceTransactionManager | Working with Spring’s JDBC abstraction support. Also useful when using iBATIS for persistence. |
| jms.connection. JmsTransactionManager | Using JMS 1.1+. |
| jms.connection. JmsTransactionManager102 | Using JMS 1.0.2. |
| orm.hibernate3. HibernateTransactionManager | Using Hibernate 3 for persistence. |
| orm.jdo.JdoTransactionManager | Using JDO for persistence. |
| orm.jpa.JpaTransactionManager | Using the Java Persistence API (JPA) for persistence. |
| transaction.jta. JtaTransactionManager | You need distributed transactions or when no other transaction manager fits the need |
| transaction.jta. OC4JJtaTransactionManager | Using Oracle’s OC4J JEE container |
| transaction.jta. WebLogicJtaTransactionManager | You need distributed transactions and your application is running within WebLogic. |
| transaction.jta. WebSphereUowTransactionManager | You need transactions managed by a UOWManager in WebSphere. |

**Java persistence API transactions**

JPATransactionManager needs to be used to coordinate the transactions.

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"> <property name="entityManagerFactory" ref="entityManagerFactory" /> </bean>

JpaTransactionManager only needs to be wired with a JPA entity manager factory

In addition to applying transactions to JPA operations, JpaTransactionManager also supports transactions on simple JDBC operations on the same DataSource used by EntityManagerFactory. For this to work, JpaTransactionManager must also be wired with an implementation of JpaDialect. For example, suppose that you’ve configured EclipseLinkJpaDialect as follows:

<bean id="jpaDialect" class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect"/>

Then you must wire the jpaDialect bean into the JpaTransactionManager like this:

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager"> <propertyname="entityManagerFactory"ref="entityManagerFactory"/> <propertyname="jpaDialect"ref="jpaDialect"/> </bean>

**Declarative transactions**

Specify Transaction annotation on class level or method level on service classes or dao classes and mention <tx:annotation-driven > element in spring application context configuration file.

**Transaction attributes**

Declarative transactions are defined with transaction attributes. Transaction attribute defines how transaction policies should be applied to method.

**Propagation behaviour**

Propagation defines the boundaries of transaction with respect to client and method being called.

|  |  |
| --- | --- |
| **Propagation behaviour** | **What it means** |
| PROPAGATION\_MANDATORY | The method must run with in transaction. If there is no existing transaction exception will be thrown. |
| PROPAGATION\_NESTED | The method should run with in nested transction if existing transaction is in progress. If there is no existing transaction it behaves like PROPAGATION\_REQUIRED. |
| PROPAGATION\_NEVER | The method should not run with in transaction. Exception will be thrown if there is an existing transaction. |
| PROPPAGATION\_NOT\_SUPPORTED | The method should not run with in transaction. If there is an existing transaction, It will be suspended during execution of this method. |
| PROPAGATION\_REQUIRED | The method should run with in transaction, If there is an existing transaction, method will be executed under it otherwise new transaction will be created. |
| PROPAGATION\_REQUIRED\_NEW | The method must run within its own trans- action. A new transaction is started and if an existing transaction is in progress, it’ll be suspended for the duration of the method. |
| PROPAGATION\_SUPPORTS | The method doesn’t require a transactional context, but may run within a transaction if one is already in progress. |

**Isolation levels**

Defines how transaction impacted by the activities of other transactions.

|  |  |
| --- | --- |
| **Isolation level** | **What it means** |
| ISOLATION\_DEFAULT | Use the default isolation level of the underlying data store |
| ISOLATION\_READ\_UNCOMMITTED | Allows you to read changes that haven’t yet been committed. May result in dirty reads, phantom reads, and nonrepeatable reads. |
| ISOLATION\_READ\_COMMITTED | Allows reads from concurrent transactions that have been com- mitted. Dirty reads are prevented, but phantom and nonrepeat- able reads may still occur. |
| ISOLATION\_REPEATABLE\_READ | Multiple reads of the same field will yield the same results, unless changed by the transaction itself. Dirty reads and nonre- peatable reads are prevented, but phantom reads may still occur. |
| ISOLATION\_SERIALIZABLE | This fully ACID-compliant isolation level ensures that dirty reads, nonrepeatable reads, and phantom reads are all pre- vented. This is the slowest of all isolation levels because it’s typically accomplished by doing full table locks on the tables involved in the transaction. |

**Read-only**

It will be used to make transaction as read-only transaction. The operations under read-only transactions should not.

**Transaction timeout**

Once timeout reached transaction will be roll back automatically.

**Rollback Rules**

Rules for roll back the operations. By default transaction are roll back on runtime exceptions but not on checked exceptions. But we can declare that a transaction can be roll back for specific checked exceptions as well as runtime exceptions.