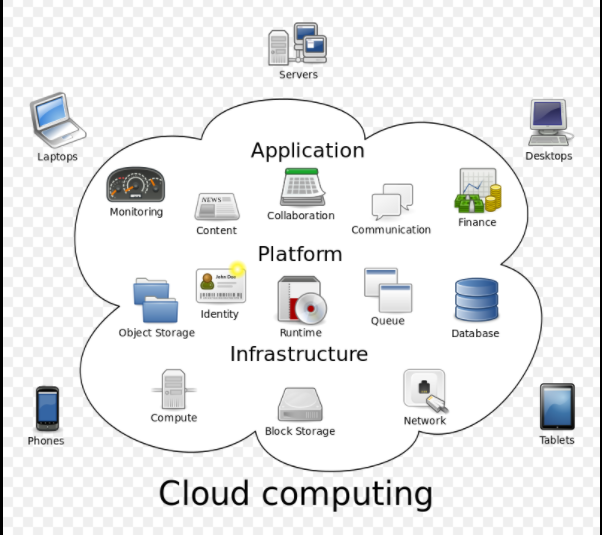
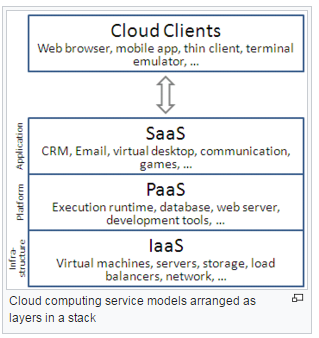
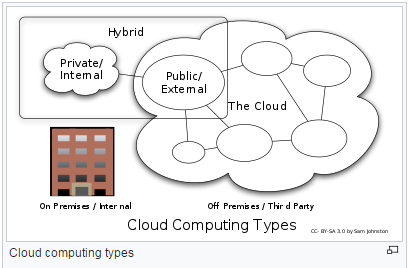
# Study Material

# **CLOUD COMPUTING**

1. **Cloud computing** is a type of [Internet](https://en.wikipedia.org/wiki/Internet)-based computing that **provides shared computer processing resources and data** **to computers and other devices** on demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources (e.g., computer networks, servers, storage, applications and services), which can be rapidly provisioned and released with minimal management effort. Cloud computing and storage solutions provide users and enterprises with various capabilities to store and process their data in either privately owned, or third-party [data centers](https://en.wikipedia.org/wiki/Data_center) that may be located far from the user–ranging in distance from across a city to across the world. Cloud computing relies on sharing of resources to achieve coherence and [economy of scale](https://en.wikipedia.org/wiki/Economies_of_scale), similar to a utility (like the [electricity grid](https://en.wikipedia.org/wiki/Electrical_grid)) over an electricity network.
2. **Characteristics:**
   1. Agility for organizations may be improved
   2. Cost reductions are claimed by cloud providers
   3. [Device and location independence](https://en.wikipedia.org/wiki/Device_independence) enable users to access systems using a web browser
   4. [Maintenance](https://en.wikipedia.org/wiki/Software_maintenance) of cloud computing applications is easier
   5. [Performance](https://en.wikipedia.org/wiki/Computer_performance) is monitored by IT experts from the service provider,
   6. [Productivity](https://en.wikipedia.org/wiki/Productivity) may be increased when multiple users can work on the same data simultaneously
3. **Service Models**: Cloud-computing providers offer their "services" according to different models, of which the three standard models per [NIST](https://en.wikipedia.org/wiki/NIST) (National Institute of Standards and Technology) are
   1. **Software as a Service (SaaS):** The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer can deploy and run arbitrary software, which can include operating systems and applications.
   2. **Platform as a Service (PaaS)**: The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.
   3. **Infrastructure as a Service (IaaS):** The capability provided to the consumer is to use the provider’s applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface.

**Also, Fourth Service Model has been integrated as**

* 1. **Mobile “backend” as a service (MBaaS)** : In the mobile "backend" as a service (m) model, also known as backend as a service (BaaS), [web app](https://en.wikipedia.org/wiki/Web_app) and [mobile app](https://en.wikipedia.org/wiki/Mobile_app) developers are provided with a way to link their applications to [cloud storage](https://en.wikipedia.org/wiki/Cloud_storage) and cloud computing services with [application programming interfaces](https://en.wikipedia.org/wiki/Application_programming_interface) (APIs) exposed to their applications and custom [software development kits](https://en.wikipedia.org/wiki/Software_development_kit) (SDKs). Services include user management, [push notifications](https://en.wikipedia.org/wiki/Push_technology), integration with [social networking services](https://en.wikipedia.org/wiki/Social_networking_service) and more. This is a relatively recent model in cloud computing, with most BaaS start-ups dating from 2011 or later but trends indicate that these services are gaining significant mainstream traction with enterprise consumers.

1. **Deployment Model**: There are mainly three type of cloud model namely:
   1. **Private Cloud**:
   2. **Public Cloud**: public cloud service providers like [Amazon Web Services](https://en.wikipedia.org/wiki/Amazon_Web_Services) (AWS), Microsoft and Google.
   3. **Hybrid Cloud**:
   4. **Others are Community cloud, Distributed Cloud etc**.
2. **Security and Privacy**: As Per the [Cloud Security Alliance](https://en.wikipedia.org/wiki/Cloud_Security_Alliance), the top three threats in the cloud are
   1. *Insecure Interfaces and API's*,
   2. *Data Loss & Leakage*, and
   3. *Hardware Failure*—which accounted for 29%, 25% and 10% of all cloud security outages respectively.

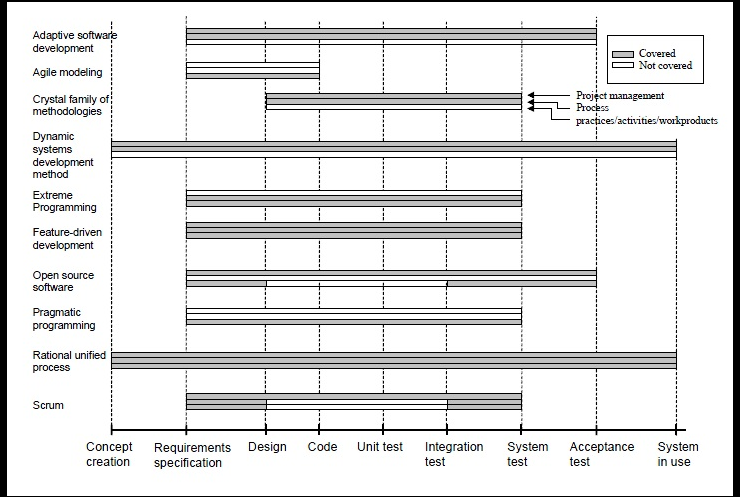
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# **Agile Software Development**

(OR Agile Principles and SCRUM)

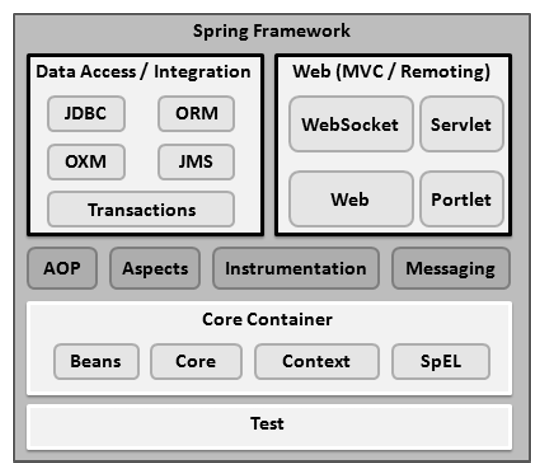
1. **Agile software development**: It describes a set of principles for [software development](https://en.wikipedia.org/wiki/Software_development) under which requirements and solutions evolve through the collaborative effort of self-organizing [cross-functional teams](https://en.wikipedia.org/wiki/Cross-functional_team). It advocates adaptive planning, evolutionary development, early delivery, and continuous improvement, and it encourages rapid and flexible response to change. These principles support the definition and continuing evolution of many [software development methods](https://en.wikipedia.org/wiki/Software_development_methodologies).
2. **Agile Principles:** The Manifesto for Agile Software Development is based on twelve principles:
3. Customer satisfaction by early and continuous delivery of valuable software
4. Welcome changing requirements, even in late development
5. **Working software is delivered frequently (weeks rather than months)**
6. Close, daily cooperation between business people and developers
7. Projects are built around motivated individuals, who should be trusted
8. Face-to-face conversation is the best form of communication (co-location)
9. Working software is the principal measure of progress
10. Sustainable development, able to maintain a constant pace
11. **Continuous attention to technical excellence and good design**
12. Simplicity—the art of maximizing the amount of work not done—is essential
13. Best architectures, requirements, and designs emerge from self-organizing teams
14. Regularly, the team reflects on how to become more effective, and adjusts accordingly
15. **Agile Overview**: **Iterative, incremental and evolutionary**

* Most agile development methods break product development work into small increments that minimize the amount of up-front planning and design. Iterations are short time frames that typically last from one to four weeks.
* Each iteration involves a [cross-functional team](https://en.wikipedia.org/wiki/Cross-functional_team) working in all functions:
* [**planning**](https://en.wikipedia.org/wiki/Project_planning)**,**[**analysis**](https://en.wikipedia.org/wiki/Requirements_analysis)**,**[**design**](https://en.wikipedia.org/wiki/Software_design)**,**[**coding**](https://en.wikipedia.org/wiki/Computer_programming)**,**[**unit testing**](https://en.wikipedia.org/wiki/Unit_testing)**, and**[**acceptance testing**](https://en.wikipedia.org/wiki/Acceptance_testing).
* At the end of the iteration a working product is demonstrated to stakeholders. T
* his minimizes overall risk and allows the product to adapt to changes quickly.
* An iteration might not add enough functionality to warrant a market release, but the goal is to have an available release (with minimal [bugs](https://en.wikipedia.org/wiki/Software_bug)) at the end of each iteration.
* Multiple iterations might be required to release a product or new features.
* Working software is the primary measure of progress.

1. **Agile Methods** : Agile software development methods support a broad range of the [software development life cycle](https://en.wikipedia.org/wiki/Software_development_life_cycle). Some focus on the practices (e.g., XP, pragmatic programming, agile modelling), while some focus on managing the flow of work (e.g., Scrum, Kanban).
2. **Agile Practices**: Agile software development is supported by several concrete practices, covering areas like r**equirements, design, modelling, coding, testing, planning, risk management, process, quality,** etc. Some notable agile software development practices include:
   * [Acceptance test-driven development](https://en.wikipedia.org/wiki/Acceptance_test-driven_development) (ATDD), [Agile modelling](https://en.wikipedia.org/wiki/Agile_modeling), [Agile testing](https://en.wikipedia.org/wiki/Agile_testing)
   * [Backlogs](https://en.wikipedia.org/wiki/Scrum_(development)#Product_Backlog) (Product and Sprint), [Behaviour-driven development](https://en.wikipedia.org/wiki/Behavior-driven_development) (BDD)
   * Business analyst designer method (BADM),[Continuous integration](https://en.wikipedia.org/wiki/Continuous_integration) (CI)
   * [Cross-functional team](https://en.wikipedia.org/wiki/Cross-functional_team), [Domain-driven design](https://en.wikipedia.org/wiki/Domain-driven_design) (DDD)
   * Information radiators (scrum board, task board, visual management board, [burndown chart](https://en.wikipedia.org/wiki/Burndown_chart))
   * [Iterative and incremental development](https://en.wikipedia.org/wiki/Iterative_and_incremental_development) (IID),[Pair programming](https://en.wikipedia.org/wiki/Pair_programming), [Planning poker](https://en.wikipedia.org/wiki/Planning_poker)
   * [Refactoring](https://en.wikipedia.org/wiki/Refactoring), [Retrospective](https://en.wikipedia.org/wiki/Retrospective)
   * [**Scrum events**](https://en.wikipedia.org/wiki/Scrum_(development))**(sprint planning, daily scrum, sprint review and retrospective)**
   * [Story-driven modelling](https://en.wikipedia.org/wiki/Story-driven_modeling), [Test-driven development](https://en.wikipedia.org/wiki/Test-driven_development) (TDD)
   * [Timeboxing](https://en.wikipedia.org/wiki/Timeboxing), [User story](https://en.wikipedia.org/wiki/User_story), User story mapping, [Velocity tracking](https://en.wikipedia.org/wiki/Velocity_(software_development))
   * The Agile Alliance has provided a comprehensive online guide to applying agile these and other practices

# **Spring Framework Learning (Version 4.1.x)**

**https://www.tutorialspoint.com/spring/spring\_architecture.htm**



1. The Spring Framework provides about 20 modules which can be used based on an application requirement.

**Core**: Core, bean, context, SpEL

**Web MVC**: Web (provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context.), Web-MVC (contains Spring's Model-View-Controller (MVC) implementation), Servlet, Portlet, WebSocket.

**Data Access**: JDBC, JMS, Transactions & OXM (supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream), ORM (provides integration layers for popular object-relational mapping APIs, including JPA, JDO, Hibernate, and iBatis)

**Miscellaneous**: AOP (define method-interceptors and pointcuts to cleanly decouple code that implements functionality that should be separated.), Aspects (provides integration with AspectJ, which is again a powerful and mature AOP framework), Messaging (provides support for STOMP as the WebSocket sub-protocol to use in applications.) and Test (testing of Spring components with JUnit).

1. IoC (Inversion of Control):The Spring **container** is at the core of the Spring Framework. The container will create the objects, wire them together, configure them, and manage their complete life cycle from creation till destruction. The Spring container uses DI to manage the components that make up an application.
   1. The container gets its instructions on what objects to instantiate, configure, and assemble by reading the configuration metadata provided.
   2. The configuration metadata can be represented either by XML, Java annotations, or Java code.
   3. The Spring **IoC** container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
   4. Two distinct types of containers **BeanFactory** container (This is the simplest container providing the basic support for DI) and **ApplicationContext** container (this container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners)
2. **Beans (its Inheritance and Template)**: The objects that form the backbone of your application and that are managed by the Spring IoC container are called beans. *You may* ***inherit*** *a bean and create the* ***template*** *which can be used in other child bean definitions*.

|  |  |  |
| --- | --- | --- |
| **No.** | **Properties** | **Description** |
| 1 | class | This attribute is mandatory and specifies the bean class to be used to create the bean. |
| 2 | name/id | This attribute specifies the bean identifier uniquely |
| 3 | scope | singleton (only one instance per IoC Container), prototype (multiple object instances), request, session, global-session |
| 4 | constructor-arg | This is used to inject the dependencies. |
| 5 | properties | This is used to inject the dependencies. |
| 6 | autowiring mode | This is used to inject the dependencies. |
| 7 | lazy-initialization | A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at the startup. |
| 8 | initialization method | A callback to be called just after all necessary properties on the bean have been set by the container. |
| 9 | destruction method | A callback to be used when the container containing the bean is destroyed. |

<beans xmlns = "http://www.springframework.org/schema/beans"

xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation = "http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<!-- A simple bean definition -->

<bean id = "..." class = "...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with scope, lazy-init, init and destroy set on -->

<bean id = "..." class = "..." scope=”singleton” lazy-init="true" init-method= "init" destroy-method="destroy" >

<!-- collaborators and configuration for this bean go here -->

</bean>

</beans>

1. **Dependency Injection**: DI (or sometime called wiring) helps in gluing these classes together and at the same time keeping them independent.

|  |  |  |
| --- | --- | --- |
| **No.** | **Type** | **Description** |
| 1 | constructor-based dependency | Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on the other class. |
| 2 | setter-based dependency | Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. |

**Note**: You can mix both, Constructor-based and Setter-based DI but it is a good rule of thumb to use constructor arguments for **mandatory** dependencies and setters for **optional** dependencies.

<!-- DI example of setter based (Dependent object could be Primitive or Object) -->

**<bean id="obj" class="com.javatpoint.Employee">**

**<property name="id" value="22"/>**

**<property name="name" value="Farkalit"/>**

**<property name="city" value="Jamia Nagar"/>**

**</bean>**

<!—Constructor based -->

**<bean id="address" class="com.farkalit.tutorial.spring.Address">**

**<constructor-arg value="Shaheen Bagh"></constructor-arg>**

**<constructor-arg value="Delhi"></constructor-arg>**

**<constructor-arg value="India"></constructor-arg>**

**</bean>**

**<bean id="employObj" class="com.farkalit.tutorial.spring.EmployeeDI">**

**<constructor-arg value="12" type="int"/>**

**<constructor-arg value="Usman"/>**

**<constructor-arg ref="address"/>**

**</bean>**

1. **Dependency Injection for Collection**: Now what if you want to pass plural values like Java Collection types such as List, Set, Map, and Properties. To handle the situation, Spring offers four types of collection configuration elements which are as follows:
   1. **List**
   2. **Set**
   3. **Map**
   4. **Props**

**<bean id = "diCollection" class = "com.farkalit.tutorial.spring.DICollection">**

**<!-- results in a setAddressList(java.util.List) call -->**

**<property name = "addressList">**

**<list>**

**<value>INDIA</value>**

**<value>Pakistan</value>**

**<value>KSA</value>**

**<value>USA</value>**

**</list>**

**</property>**

**</bean>**

1. **DI Auto-wiring:** The Spring container can auto-wire relationships between collaborating beans without using <constructor-arg> and <property> elements, which helps cut down on the amount of XML configuration you write for a big Spring-based application.

|  |  |  |
| --- | --- | --- |
| No. | Type | Description |
| 1 | no | This is default setting which means no autowiring. |
| 2 | byName | Autowiring by property name. Spring container looks at the properties of the beans on which autowire attribute is set to byName in the XML configuration file. |
| 3 | byType | Autowiring by property datatype. Spring container looks at the properties of the beans on which autowire attribute is set to byType in the XML configuration file. |
| 4 | constructor | Similar to byType, but type applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised. |
| 5 | autodetect | Spring first tries to wire using autowire by constructor, if it does not work, Spring tries to autowire by byType. |

**Limitations with autowiring**: You cannot autowire so-called simple properties such as primitives, Strings, and Classes.

Two way to do autowiring:

1. Add Spring context and <context:annotation-config /> in bean configuration file.
2. Include ‘AutowiredAnnotationBeanPostProcessor’ directly in bean configuration file.

public class Customer {

private Person person;

private int type;

private String action;

//Autowired setter method

@Autowired

public void setPerson(Person person) {

this.person = person;

}

//Autowired constructor method

// @Autowired

// public Customer(Person person) {

// this.person = person;

// }

//Autowired field

// @Autowired

// private Person person;

}

<!-- Auto wiring Dependency Injection -->

<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor"/>

<bean id="customerBean" class="com.farkalit.tutorial.spring.autowire.Customer">

<property name="action" value="buy" />

<property name="type" value="1" />

</bean>

<bean id="personBean" class="com.farkalit.tutorial.spring.autowire.Person">

<property name="name" value="Farkalit Usman" />

<property name="address" value="D-80 Jamia Nagar" />

<property name="age" value="29" />

</bean>

1. **Annotation Based Configuration**: Annotation injection is performed before XML injection. Thus, the latter configuration will override the former for properties wired through both approaches.

Before we can use annotation-based wiring, we will need to enable it in our Spring configuration file.

<context:annotation-config/>

Once <context:annotation-config/> is configured, you can start annotating your code to indicate that Spring should automatically wire values into properties, methods, and constructors.

@**Required**: This annotation applies to bean property setter methods.

@**Autowired**: This annotation can apply to bean property setter methods, non-setter methods, constructor and properties.

@**Qualifier**: This annotation along with @Autowired can be used to remove the confusion by specifying which exact bean will be wired.

**JSR-250 Annotations**: It include @Resource, @PostConstruct and @PreDestroy annotations

1. **Aspect Oriented Programming**: Aspect-Oriented Programming entails breaking down program logic into distinct parts called so-called concerns. The functions that span multiple points of an application are called cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic. There are various common good examples of aspects like logging, auditing, declarative transactions, security, caching, etc.

|  |  |  |
| --- | --- | --- |
| **No.** | **Properties** | **Description** |
| 1 | Aspect | This is a module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement. |
| 2 | Join Point | This represents a point in your application where you can plug-in the AOP aspect. You can also say; it is the actual place in the application where an action will be taken using Spring AOP framework. |
| 3 | Advice | This is the actual action to be taken either before or after the method execution. This is an actual piece of code that is invoked during the program execution by Spring AOP framework. |
| 4 | Pointcut | This is a set of one or more join points where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples. |
| 5 | Introduction | An introduction allows you to add new methods or attributes to the existing classes. |
| 6 | Target Object | The object being advised by one or more aspects. |
| 7 | Weaving | Weaving is the process of linking aspects with other application types or objects to create an advised object |

1. **AOP Advice:** Spring aspects can work with five kinds of advice mentioned as follows

|  |  |  |
| --- | --- | --- |
| **No.** | **Properties** | **Description** |
| 1 | before | Run advice before the method execution. |
| 2 | after | Run advice after the method execution, regardless of its outcome. |
| 3 | after-returning | Run advice after the method execution only if method completes successfully. |
| 4 | after-throwing | Run advice after the method execution only if method exits by throwing an exception. |
| 5 | around | Run advice before and after the advised method is invoked. |

1. **Custom Aspects Implementation**: Spring supports the @**AspectJ** annotation style approach and the **schema-based** approach to implement custom aspects.

**Example of schema**-based:

<aop:config>

<aop:pointcut id = "selectAll" expression = "execution(\* com.farkalit.tutorial.spring.aop.Student.\*(..))"/>

<aop:aspect id = "log" ref = "logging">

<aop:before pointcut-ref = "selectAll" method = "beforeAdvice"/>

<aop:after pointcut-ref = "selectAll" method = "afterAdvice"/>

<aop:after-returning pointcut-ref = "selectAll" returning = "retVal" method = "afterReturningAdvice"/>

<aop:after-throwing pointcut-ref = "selectAll" throwing = "ex" method = "afterThrowingAdvice"/>

</aop:aspect>

</aop:config>

<!-- Definition for student bean -->

<bean id = "student" class = "com.farkalit.tutorial.spring.aop.Student">

<property name = "name" value = "Usman" />

<property name = "age" value = "11"/>

</bean>

<!-- Definition for logging aspect -->

<bean id = "logging" class = "com.farkalit.tutorial.spring.aop.Logging"/>

1. **Spring JDBC Framework**: Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.

**The JDBC Template** class executes SQL queries, updates statements, stores procedure calls, performs iteration over ResultSets, and extracts returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JdbcTemplate class are thread safe once configured.

The DAO support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA, or JDO in a consistent way.

Now we need to supply a DataSource to the JDBC Template so it can configure itself to get database access.

<bean id = "dataSource"

class = "org.springframework.jdbc.datasource.DriverManagerDataSource">

<property name = "driverClassName" value = "com.mysql.jdbc.Driver"/>

<property name = "url" value = "jdbc:mysql://localhost:3306/TEST"/>

<property name = "username" value = "root"/>

<property name = "password" value = "password"/>

</bean>

<!-- Definition for studentJDBCTemplate bean -->

<bean id = "studentJDBC"

class = "com.tutorialspoint.StudentJDBCTemplate">

<property name = "dataSource" ref = "dataSource" />

</bean>

public class StudentJDBCTemplate implements StudentDAO {

private DataSource dataSource;

private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {

this.dataSource = dataSource;

this.jdbcTemplate = new JdbcTemplate(dataSource);

}

public void create(String name, Integer age) {

String SQL = "insert into Student (name, age) values (?, ?)";

jdbcTemplate.update( SQL, name, age);

System.out.println("Record Name = " + name + " Age = " + age);

return;

}

//…TODO -task

}

1. **Spring Transaction Management**: A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of RDBMS-oriented enterprise application to ensure data integrity and consistency. The concept of transactions can be described with the following four key properties described as **ACID (Atomicity, Consistency, Isolation, Durability)**.

* **Atomicity** − A transaction should be treated as a single unit of operation, which means either the entire sequence of operations is successful or unsuccessful.
* **Consistency** − This represents the consistency of the referential integrity of the database, unique primary keys in tables, etc.
* **Isolation** − There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.
* **Durability** − Once a transaction has completed, the results of this transaction should be made permanent and cannot be erased from the database due to system failure.

Transaction cycle is:

begin() Transaction;

Performs operation of insert, update, delete.

commit() changes on success else rollback() all the operations.

<!-- Initialization for TransactionManager -->

<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

<!-- Definition for ServiceImpl bean -->

<bean id="serviceImpl" class="com.service.ServiceImpl">

<constructor-arg ref="transactionManager"/>

</bean>

public class ServiceImpl implements Service

{

private final TransactionTemplate transactionTemplate;

// use constructor-injection to supply the PlatformTransactionManager

public ServiceImpl(PlatformTransactionManager transactionManager)

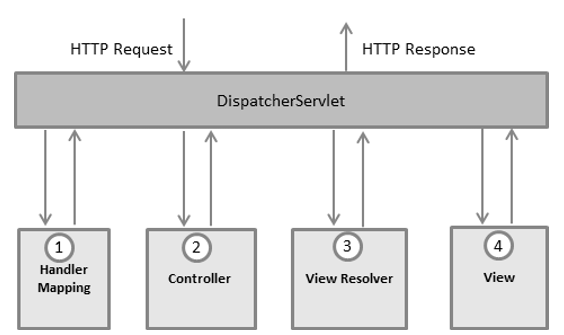
{

this.transactionTemplate = new TransactionTemplate(transactionManager);

}

//..TODO -task

}

1. **Spring MVC**: The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications.

<context:component-scan base-package = "com.farkalit.spring.controller" />

<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name = "prefix" value = "/WEB-INF/jsp/" />

<property name = "suffix" value = ".jsp" />

</bean>

1. **Difference between Spring 3.0 and Spring 4.0**

**Here are some important improvements in Spring 4.x**

* Removed Deprecated Packages and Methods, check out the API Differences Report
* Java 8 Support
* Java EE 6 and 7 or above is now considered the baseline for Spring Framework 4
* Groovy Bean Definition DSL, read more about this API.
* Core Container Improvements
* General Web Improvements
* WebSocket, SockJS, and STOMP Messaging
* Testing Improvements

**And Spring 3.x which introduced lot of new features like:**

1. Spring MVC Test Framework
2. Asynchronous MVC processing on Servlet 3.0
3. custom @Bean definition annotations in @Configuration classes
4. @Autowired and @Value to be used as meta-annotations
5. Concurrency refinements across the framework
6. loading WebApplicationContexts in the TestContext framework
7. JCache 0.5 (JSR-107)

# **Hibernate (Version 4.3.x)**

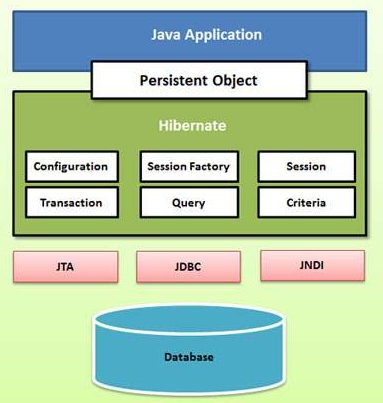
**https://www.tutorialspoint.com/hibernate/hibernate\_interceptors.htm**

1. **Hibernate** is an Object-Relational Mapping(ORM) solution for JAVA and it raised as an open source persistent framework.

An ORM solution consists of the following four entities:

* An API to perform basic CRUD operations on objects of persistent classes.
* A language or API to specify queries that refer to classes and properties of classes.
* A configurable facility for specifying mapping metadata.
* A technique to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

There are several persistent frameworks and ORM options in Java.

* Enterprise JavaBeans Entity Beans
* Java Data Objects
* Castor
* TopLink
* Spring DAO
* Hibernate
* MyBatis

Hibernate maps Java classes to database tables and from Java data types to SQL data types and relieve the developer from 95% of common data persistence related programming tasks.

Following is a detailed view of the Hibernate Application Architecture with few important core classes.

1. **Hibernate Configuration**:

All set of configuration setting information is usually supplied as a standard Java properties file called **hibernate.properties**, or as an XML file named **hibernate.cfg.xml**.

Configuration Object:

* 1. **SessionFactory** Object: The SessionFactory is a thread safe object and used by all the threads of an application. You would need one SessionFactory object per database using a separate configuration file.
  2. **Transaction** Object: This is an optional object and Hibernate applications may choose not to use this interface, instead managing transactions in their own application code.
  3. **Query** Object: A Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.
  4. **Criteria Object:** Criteria object are used to create and execute object oriented criteria queries to retrieve objects.

Example of 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>

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost/test</property>

<property name="hibernate.connection.username">root</property>

<property name="hibernate.connection.password">root</property>

<property name="hibernate.show\_sql">false</property>

<!--<property name="hbm2ddl.auto">update</property>-->

<property name="hibernate.connection.pool\_size">5</property>

<!-- List of XML mapping files for with annotaion -->

<mapping class="com.farkalit.annotation.employee.test.EmployeeInfo"/>

<mapping class="com.farkalit.annotation.user.test.User"/>

</session-factory>

</hibernate-configuration>

1. **Session** Object: From session factory object, we can get a session object. A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. **Instances (not session) may exist in one of the following three states at a given point in time:**

* **transient:** A new instance of a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
* **persistent:** You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
* **detached:** Once we close the Hibernate Session, the persistent instance will become a detached instance.

Session session = factory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

// do some work

...

tx.commit();

}

catch (Exception e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

There are number of methods provided by the Session interface. Some are here:

session.beginTransaction();

session.createCriteria(String str);

session.createFilter(Object obj, String str);

session.get(String str, Serializable ser);//return null if no data found

session.load();// throws ObjectNotFoundException if data not found.

session.save();

session.persist();

session.saveOrUpdate();

session.delete();

session.update();

session.close();

1. **Hibernate - O/R Mappings (Collections Mappings)**

If an entity or class has collection of values for a particular variable, then we can map those values using any one of the collection interfaces available in java. Hibernate can persist instances **of java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List,** and any array of persistent entities or values.

|  |  |
| --- | --- |
| Collection type | Mapping and Description |
| [**java.util.Set**](https://www.tutorialspoint.com/hibernate/hibernate_set_mapping.htm) | This is mapped with a <set> element and initialized with java.util.HashSet |
| [**java.util.SortedSet**](https://www.tutorialspoint.com/hibernate/hibernate_sortedset_mapping.htm) | This is mapped with a <set> element and initialized with java.util.TreeSet. The sort attribute can be set to either a comparator or natural ordering. |
| [**java.util.List**](https://www.tutorialspoint.com/hibernate/hibernate_list_mapping.htm) | This is mapped with a <list> element and initialized with java.util.ArrayList |
| [**java.util.Collection**](https://www.tutorialspoint.com/hibernate/hibernate_bag_mapping.htm) | This is mapped with a <bag> or <ibag> element and initialized with java.util.ArrayList |
| [**java.util.Map**](https://www.tutorialspoint.com/hibernate/hibernate_map_mapping.htm) | This is mapped with a <map> element and initialized with java.util.HashMap |
| [**java.util.SortedMap**](https://www.tutorialspoint.com/hibernate/hibernate_sortedmap_mapping.htm) | This is mapped with a <map> element and initialized with java.util.TreeMap. The sort attribute can be set to either a comparator or natural ordering. |

**Association Mappings:**

The mapping of associations between entity classes and the relationships between tables is the soul of ORM. Following are the four ways in which the cardinality of the relationship between the objects can be expressed. An association mapping can be unidirectional as well as bidirectional.

|  |  |
| --- | --- |
| Mapping type | Description |
| [**Many-to-One**](https://www.tutorialspoint.com/hibernate/hibernate_many_to_one_mapping.htm) | Mapping many-to-one relationship using Hibernate |
| [**One-to-One**](https://www.tutorialspoint.com/hibernate/hibernate_one_to_one_mapping.htm) | Mapping one-to-one relationship using Hibernate |
| [**One-to-Many**](https://www.tutorialspoint.com/hibernate/hibernate_one_to_many_mapping.htm) | Mapping one-to-many relationship using Hibernate |
| [**Many-to-Many**](https://www.tutorialspoint.com/hibernate/hibernate_many_to_many_mapping.htm) | Mapping many-to-many relationship using Hibernate |

1. **Hibernate - Annotations**

Hibernate annotations is the newest way to define mappings without a use of XML file. You can use annotations in addition to or as a replacement of XML mapping metadata.

import javax.persistence.\*;

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@Id @GeneratedValue

@Column(name = "id")

private int id;

@Column(name = "first\_name")

private String firstName;

@Column(name = "last\_name")

private String lastName;

@Column(name = "salary")

private int salary;

public Employee() {}

public int getId() {

return id;

}

//…

}

@**Column** Annotation:

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. You can use column annotation with the following most commonly used attributes:

* **name** attribute permits the name of the column to be explicitly specified.
* **length** attribute permits the size of the column used to map a value particularly for a String value.
* **nullable** attribute permits the column to be marked NOT NULL when the schema is generated.
* **unique** attribute permits the column to be marked as containing only unique values.

1. **Hibernate – Query Language**

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();

String hql = "SELECT E.firstName FROM Employee E";//select clause

String hql = "FROM Employee E WHERE E.id = 10"; // where clause

String hql = "FROM Employee E WHERE E.id > 10 " +

"ORDER BY E.firstName DESC, E.salary DESC ";//order by clause

String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +

"GROUP BY E.firstName";//group by clause

String hql = "FROM Employee E WHERE E.id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id",10);//Named Parameter; To update and delete

int result = query.executeUpdate();// For insert, update and delete

**Aggregate Methods:**

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

|  |  |  |
| --- | --- | --- |
| S.N. | Functions | Description |
| 1 | avg(property name) | The average of a property's value |
| 2 | count(property name or \*) | The number of times a property occurs in the results |
| 3 | max(property name) | The maximum value of the property values |
| 4 | min(property name) | The minimum value of the property values |
| 5 | sum(property name) | The sum total of the property values |

String hql = "SELECT count(distinct E.firstName) FROM Employee E";

**Pagination using Query:**

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **Query setFirstResult(int startPosition)**  This method takes an integer that represents the first row in your result set, starting with row 0. |
| 2 | **Query setMaxResults(int maxResult)**  This method tells Hibernate to retrieve a fixed number maxResults of objects. |

String hql = "FROM Employee";

Query query = session.createQuery(hql);

query.setFirstResult(1);

query.setMaxResults(10);

List results = query.list();

1. **Hibernate - Criteria Queries**

Hibernate provides alternate ways of manipulating objects and in turn data available in RDBMS tables. One of the methods is Criteria API which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000

cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara

cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.

cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000

cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null

cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null

cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty

cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty

cr.add(Restrictions.isNotEmpty("salary"));

You can create AND or OR conditions using LogicalExpression restrictions as follows:

Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);

Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR conditions

LogicalExpression orExp = Restrictions.or(salary, name);

cr.add( orExp );

// To get records matching with AND conditions

LogicalExpression andExp = Restrictions.and(salary, name);

cr.add( andExp );

List results = cr.list();

Note: Pagination also allowed in Criteria.

**Sorting the Results:**

Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order

crit.addOrder(Order.desc("salary"));

// To sort records in ascending order

crit.addOrder(Order.asc("salary"));

List results = cr.list();

1. **Hibernate - Projections Queries**

**Projections & Aggregations:**

The Criteria API provides the **org.hibernate.criterion.Projections** class which can be used to get average, maximum or minimum of the property values. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances.

Criteria cr = session.createCriteria(Employee.class);

// To get total row count.

cr.setProjection(Projections.rowCount());

// To get average of a property.

cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.

cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.

cr.setProjection(Projections.max("salary"));

// To get minimum of a property.

cr.setProjection(Projections.min("salary"));

// To get sum of a property.

cr.setProjection(Projections.sum("salary"));

1. **Hibernate – Native SQL**

You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

public SQLQuery createSQLQuery(String sqlString) throws HibernateException

After you pass a string containing the SQL query to the createSQLQuery() method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

**Scalar queries:**

The most basic SQL query is to get a list of scalars (values) from one or more tables. Following is the syntax for using native SQL for scalar values:

String sql = "SELECT first\_name, salary FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.setResultTransformer(Criteria.ALIAS\_TO\_ENTITY\_MAP);

List results = query.list();

**Entity queries:**

The above queries were all about returning scalar values, basically returning the "raw" values from the resultset. The following is the syntax to get entity objects as a whole from a native sql query via addEntity().

String sql = "SELECT \* FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

List results = query.list();

**Named SQL queries:**

The following is the syntax to get entity objects from a native sql query via addEntity() and using named SQL query.

String sql = "SELECT \* FROM EMPLOYEE WHERE id = :employee\_id";

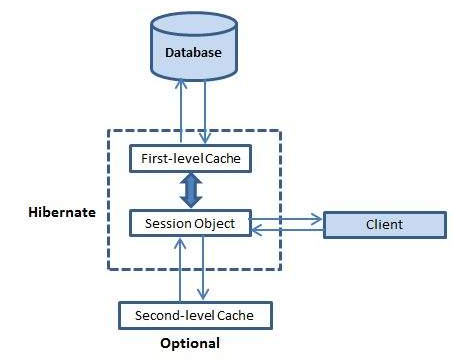
SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

query.setParameter("employee\_id", 10);

List results = query.list();

1. **Hibernate - Caching**

Caching is all about application performance optimization and it sits between your application and the database to avoid the number of database hits as many as possible to give a better performance for performance critical applications.

**First-level cache:**

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

**Second-level cache:**

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

**Concurrency strategies:**

A concurrency strategy is a mediator which responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

* **Transactional**: Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
* **Read-write**: Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
* **Nonstrict-read-write**: This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.
* **Read-only**: A concurrency strategy suitable for data which never changes. Use it for reference data only.

<hibernate-mapping>

<class name="Employee" table="EMPLOYEE">

<cache usage="read-write"/>

<id name="id" type="int" column="id">

<generator class="native"/>

</id>

<property name="firstName" column="first\_name" type="string"/>

</class>

</hibernate-mapping>

The usage="**read-write**" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

**Cache Provider:**

Hibernate forces you to choose a single cache provider for the whole application. Following are the few CACHE provider name:

* EHCache
* OSCache
* warmCache
* JBoss Cache

**Note: Vertex IVR uses cachedb to cache data into database.**

**The Query-level Cache:**

To use the query cache, you must first activate it using the **hibernate.cache.use\_query\_cache="true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

Session session = SessionFactory.openSession();

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

query.setCacheRegion("employee");

List users = query.list();

SessionFactory.closeSession();

1. **Hibernate - Batch Processing**

Consider a situation when you need to upload a large number of records into your database using Hibernate. Because by default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an **OutOfMemoryException** somewhere around the 50,000th row. You can resolve this problem if you are using batch processing with Hibernate.

To use the batch processing feature, first set hibernate.jdbc.batch\_size as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch.

<property name="hibernate.jdbc.batch\_size">50</property>

To implement this in your code we would need to do little modification as follows:

Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

Employee employee = new Employee(.....);

session.save(employee);

if( i % 50 == 0 ) { // Same as the JDBC batch size

//flush a batch of inserts and release memory:

session.flush();

session.clear();

}

}

tx.commit();

session.close();

Above code will work fine for the INSERT operation, but if you are willing to make UPDATE operation then you can achieve using the following code:

ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")

.scroll();

int count = 0;

while ( employeeCursor.next() ) {

Employee employee = (Employee) employeeCursor.get(0);

employee.updateEmployee();

seession.update(employee);

if ( ++count % 50 == 0 ) {

session.flush();

session.clear();

}

}

tx.commit();

session.close();

1. **Hibernate – Interceptors**

An object passes through different stages in its life cycle and **Interceptor** **Interface** provides methods which can be called at different stages to perform some required tasks. These methods are 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. Following is the list of all the methods available within the Interceptor interface:

|  |  |
| --- | --- |
| **S.N.** | **Method and Description** |
| 1 | **findDirty()**  This method is be called when the **flush()** method is called on a Session object. |
| 2 | **instantiate()**  This method is called when a persisted class is instantiated. |
| 3 | **isUnsaved()**  This method is called when an object is passed to the **saveOrUpdate()** method/ |
| 4 | **onDelete()**  This method is called before an object is deleted. |
| 5 | **onFlushDirty()**  This method is called when Hibernate detects that an object is dirty (ie. have been changed) during a flush i.e. update operation. |
| 6 | **onLoad()**  This method is called before an object is initialized. |
| 7 | **onSave()**  This method is called before an object is saved. |
| 8 | **postFlush()**  This method is called after a flush has occurred and an object has been updated in memory. |
| 9 | **preFlush()**  This method is called before a flush. |

To build an interceptor you can either implement **Interceptor** class directly or extend **EmptyInterceptor** class. Following will be the simple steps to use Hibernate Interceptor functionality.

public class MyInterceptor extends EmptyInterceptor {

private int updates;

private int creates;

private int loads;

//called before commit into database

public void preFlush(Iterator iterator) {

System.out.println("preFlush");

}

//called after committed into database

public void postFlush(Iterator iterator) {

System.out.println("postFlush");

}

}

**private static SessionFactory buildSessionFactory()**

**{**

**try**

**{**

**Configuration conf = new Configuration();**

**/\* \* Set the my interceptor \*/**

**conf.setInterceptor(new MyInterceptor());//Optional**

**// Create the SessionFactory from hibernate.cfg.xml**

**conf.configure("hibernate.cfg.xml");**

**return conf.buildSessionFactory();**

**}**

**catch (HibernateException ex)**

**{**

**}**

**}**

**Question Answer of Hibernate:**

**What is the difference between save() and persist() methods of session object?**

session.save saves the object and returns the id of the instance whereas persist do not return anything after saving the instance.

**What is the difference between get() and load() methods of session object?**

There are following differences between get() and load() methods.

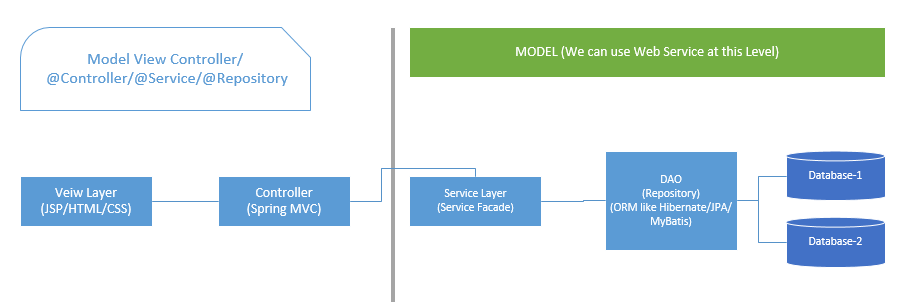
* get() returns null if no data is present where as load throws ObjectNotFoundException exception in such case.
* get() always hits the database whereas load() method doesn't hit the database.
* get() returns actual object whereas load() returns proxy object.
* A central feature of Hibernate, proxies, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.
* All classes that do not extend or implement some specialized classes and interfaces required by the EJB framework.

**What is lazy loading?**

Lazy loading is a technique in which objects are loaded on demand basis. Since Hibernate 3, lazy loading is by default, enabled so that child objects are not loaded when parent is loaded.

**Application Architecture**

------------------------------------



### What is Transaction Isolation Levels

The ANSI/ISO SQL standard defines four levels of transaction isolation, with different possible outcomes for the same transaction scenario. That is, the same work performed in the same fashion with the same inputs may result in different answers, depending on your isolation level. These levels are defined in terms of three phenomena that are either permitted or not at a given isolation level:

* **Dirty read:** The meaning of this term is as bad as it sounds. You're permitted to read uncommitted, or dirty, data. You can achieve this effect by just opening an OS file that someone else is writing and reading whatever data happens to be there. Data integrity is compromised, foreign keys are violated, and unique constraints are ignored.
* **Nonrepeatable read:**This simply means that if you read a row at time T1 and try to reread that row at time T2, the row may have changed. It may have disappeared, it may have been updated, and so on.
* **Phantom read:**This means that if you execute a query at time T1 and re-execute it at time T2, additional rows may have been added to the database, which may affect your results. This differs from a nonrepeatable read in that with a phantom read, data you already read hasn't been changed, but instead, more data satisfies your query criteria than before.

Note that the ANSI/ISO SQL standard defines transaction-level characteristics, not just individual statement-by-statement-level characteristics. I'll examine transaction-level isolation, not just statement-level isolation.

The SQL isolation levels are defined based on whether they allow each of the preceding phenomena. It's interesting to note that the SQL standard doesn't impose a specific locking scheme or mandate particular behaviors, but rather describes these isolation levels in terms of these phenomena—allowing for many different locking/concurrency mechanisms to exist (see Table 1).

|  |  |  |  |
| --- | --- | --- | --- |
| Isolation Level | Dirty Read | Nonrepeatable Read | Phantom Read |
| READ UNCOMMITTED | Permitted | Permitted | Permitted |
| READ COMMITTED | -- | Permitted | Permitted |
| REPEATABLE READ | -- | -- | Permitted |
| SERIALIZABLE | -- | -- | -- |

|  |
| --- |
|  |
| Table 1: ANSI isolation levels |