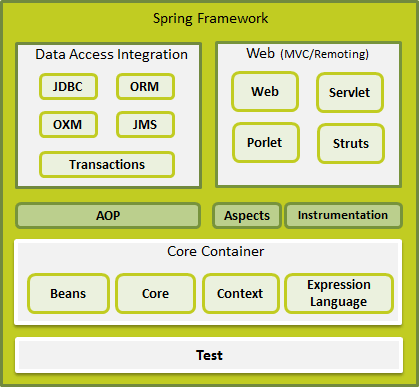
**SPRING FRAMEWORK**

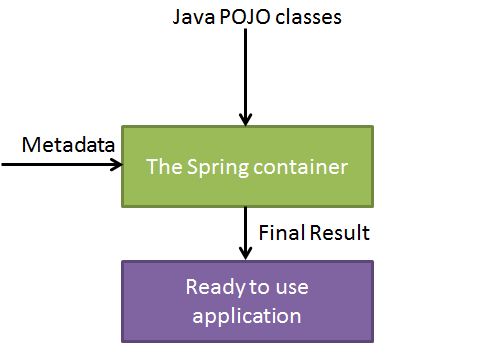
**Spring Architecture:**

* Provides about 20 modules which can be based on an application requirement.

****

Spring Architecture

* IoC(Inversion of Control) containers:
  + Makes use of Java POJO(Plain Old Java Object) classes and configuration metadata (ex: applicationContext.xml,…) to produce a fully configured and executable system or application.

****

* + BeanFactory: usually use for light weight applications like mobile devices or applet…
  + ApplicationContext: includes all functionality of the BeanFactory container.

**Spring Bean Definition**

* A bean is an object (a class), is created with the configuration metadata that you supply to the container ( ex: in applicationContext.xml: <bean/> )
* Properties of Bean:
  + Class (class=”\*\*\*”): map to the class to be used create the bean.
  + Name (id=”\*\*\*”): is the bean identifier uniquely.
  + Scope: update later (UL)
  + Constructor-arg: UL
  + Properties: UL
  + Autowiring mode: U:
  + Lazy-initialization mode (lazy-init=”\*\*\*”): create a bean instance when it is first requested, rather than at startup.
  + Initialization method (init-method=”\*\*\*”): a callback (method) to be called just after all necessary properties on the bean have been set by the container.
  + Destruction method (destroy-method=”\*\*\*”): a callback to be used when the container containing the bean is destroyed.
* Spring Configuration Metadata
  + XML configuration file (ex: applicationContext.xml,…)
  + Annotation configuration.
  + Java configuration.

**Spring bean scopes**

* Ex: to force Spring to produce a new bean instance each time one is needed, you should declare the bean’s scope attribute to be **prototype**. Similar way if you want Spring to return the same bean instance each time one is needed, you should declare the bean’s scope attribute to be **singleton**.
* Supports five scopes, three of which are available only if you use a web-aware ApplicationContext.
  + Singleton: this scopes the bean definition to a single instance per Spring IoC container (default).
  + Prototype: this scopes a single bean definition to have any number of object instances.
  + Request: this scopes a bean definition to an HTTP request. (Only valid in ApplicationContext.mxl).
  + Session: this scopes a bean definition to an HTTP seesion (Only valid…)
  + Global-session: this scopes a bean definition to a global HTTP session. (Only valid…)
* The singleton scope:
  + IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.
  + The default scope is always singleton however, when you need one and only one instance of a bean, you can set the scope property to singleton in the bean configuration file.
* The prototype scope:
  + IoC container creates new bean instance of the object every time a request for that specific bean is made.
  + Rule: use the prototype scope for all state-full beans and the singleton scope for stateless beans.

**Spring bean life cycle**

* When a bean is instantiated or is no longer required and is removed from the container, some cleanup (or uhm, action) may be required.
* Important: the <bean> with **init-method** and/or **destroy-method**.
  + Init-method: a method that is to be called on the bean immediately upon instantiation.
  + Destroy-method: a method that is called just before a bean is removed from the container.

**Spring Dependency Injection**

* In a complex Java application, app classes should be as independent as possible of other Java classes to increase the possibility to reuse these classes and to test them independently of other classes while doing unit testing. DI (or sometime called wiring) helps in gluing these classes together and same time keeping them independent.
* DI Type
  + Constructor DI: is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.
  + Setter 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.
* Can mix both DI Type but it is a good rule to use: Constructor for mandatory dependencies and Setters for optional dependencies.

**Spring injecting inner beans**

* Inner beans are beans that are defined within the scope of another bean. Inside <property/> or <constructor-arg/>.

**Spring injecting Collection**

* <list>; <set>; <map>; <props>

**Spring beans autowiring**

**Spring Annotation Based Configuration (like beans autowiring)**

* @Required: applied to bean property setter methods.
* @Autowired: cap apply to bean property setter methods, non-setter methods, constructor and properties.
* @Qualifier: can be used to remove the confusion by specifying which exact bean will be wired.
* @Resource
* @PostConstruct
* @PreDestroy

**Spring Java based Configuration**

* @Configuration
* @Bean

**Spring Event Handling**

* Event handling in the ApplicationContext is provide through the ApplicationEvent class and ApplicationListener interface. So if a bean implements the ApplicationListener, then every time an ApplicationEvent gets publisched to the ApplicationContext, that bean is notified.
* **ContextRefreshedEvent**: this event is published when the ApplicationContext is either initialized or refreshed. Can using the refresh() method on the ConfigurableApplicationContext interface.
* **ContextStartedEvent**: is published when the ApplicationContext is started using the start() method on the ConfigurableApplicationContext interface. Can poll your database or you can re/start any stopped application after receiving this event.
* **ContextStoppedEvent**: is published when the ApplicationContext is stopped using the stop() method on the ConfigurableApplicationContext interface. Can do required housekeep work after receiving this event.
* **ContextCloseEvent**: is published when the ApplicationContext is closed using the close() method on the ConfigurableApplicationContext interface. A closed context reaches its end of life; it cannot be refreshed or restarted.
* **RequestHandledEvent**: this is a web-specific event telling all beans that an HTTP request has been serviced.

**Spring Framework – AOP (Aspect oriented programming)**

* Good example of aspects like: logging, auditing, declarative transactions, security, caching, etc.
* @Aspect: declares the class as aspect.
* @Pointcut: declares the pointcut expression.
* @Before: run advice before the method execution.
* @After: run advice after the method execution regardless of its outcome.
* @AfterReturning: run advice after the method execution only if method completes successfully.
* @AfterThrowing: run advice after the method execution only if method exits by throwing an exception.
* @Around: run advice before and after the advised method is invoked.

**Spring JDBC Framework**

* Spring JDBC Templete: provides methods to write the queries directly, save lots of work and time.
  + JDBC – Java DataBase Connectivity
  + Method of Spring JDBC Template:
    - Public int update(String sql) : insert,update, delete records.
    - Public int update(String sql, Object…args): insert, update, delete using PreparedStatement given arguments.
    - Public void execute(String query) : to execute query.
    - Public T execute(Spring sql, PreparedStatementCallback action): execute query by PreparedStatement callback.
    - Public T query(String sql, ResultSetExtractor rse): fetch records using ResultsSetExtractor.
    - Public List query(String sql, RowMapper rse): fetch records using RowMapper.
  + PreparedStatement: to use execute() method with parameters query
    - Public T execute(String sql, PreparedStatementCallback<T>){\*};
    - PreparedStatementCallback: processes the input parameterized and output results.
    - Only one method:
      * \*Public T doInPreparedStatement(PreparedStatement ps) throws SQLException, DataAccessException{}
  + RowMapper:
    - Public T query(String sql, RowMapper<T> rm){\*}
    - \*Public T mapRow(ResultSet rs, int rowNumber) throws SQLException{}
  + SimpleJDBCTemplete

**Spring Transaction Management**

**Spring MVC Framework**

****

MVC

* **Model**: encapsulates the application data and in general they will consist of POJO.
* **View**: is responsible for rendering the model data and in general it generates HTML output that the client’s browser can interpret.
* **Controller**: is responsible for processing user requests and building appropriate model and passes it to the view for rendering.
* The DispatcherServlet(DS): handles all the HTTP requests and responses.
  + After receiving an HTTP request, DS > HandlerMapping > call the appropriate Controller.
  + Controller takes requests >> call service methods based on used GET or POST >> service method will set model data based on business logic and returns view name to the DispatcherServlet.
  + DS will take help from ViewResolver to pickup the defined view for the request.
  + Once view is finalized, DS passes the model data to the view which is finally rendered on the browser.
  + @Controller: this annotation marks a class as Controller.
  + @RequestMapping: this annotation map a request from client to a method ( or class ) tương ứng.
  + Web.xml: config all requests will be forwarded to the DispatcherServlet.
  + Define the bean in xml file to work with DispatcherServlet:
    - <context:component-scan>: where DispatcherServlet will search controller class.
    - Bean class InternalResourceViewResolver: to work with jsp page.

**Spring MVC Tiles**

* chia layout thành nhiều phần riêng biệt, dễ dàng cập nhật từng phần riêng biệt.

**Virtual-Host Apache Tomcat**

* Create virtual host on local & virtual machine with Tomcat.
* Configure file hosts: hostname with ipaddress from windows/system32/driver/etc/hosts(local) & /etc/hosts(virtual machine)
* Configure server.xml in Tomcat: to create virtual host, create different hostname with the same ipaddress.

**RESTeasy**

* REST: Representational State Transfer
* Annotation Support In REST
  + @Path: path for a class or method.
  + @GET: get (select) resource from server.
  + @POST: create new resource
  + @DELETE: delete resource
  + @PUT: update resource.
  + @Produces: It specifies the MIME media type of the response of a resource.
  + @Consumes: It specifies the MIME media type of the request to a resource.
  + @PathParam: to extract a parameter from the uri path of resource.
  + @QueryParam: to extract a parameter form the uri path of resource.

**Maven Framework**

* Apache Maven is a build tool to support the developer at the whole process of a software project.
* Features of Maven:
  + Convention over configuration
  + Dependency management
  + Central repository
  + Extensible via plug-ins

**HIBERNATE FRAMEWORK**

**Hibernate Framework**

* To interact the database with java application. Hibernate is an open source, lightweight, ORM (Object Relational Mapping) tool which simplifies the data creation, data manipulation and data access. It is a programming technique that maps the object to the data stored in the database.



**Advantages of Hibernate Framework**

* Opensource and lightweight.
* Fast performance: because cache ( 1st cache-default & 2nd cache) is internally used in hibernate framework.
* Database Independent query: HQL (Hibernate Query Language) generates the database independent queries.
* Automatic table creation.
* Simplifies complex join: fetch data from multiple tables.
* Provides query statistics and database status.

**Hibernate Architecture**



Hibernate Architecture with mapping file and configuration file.

* Mapping file ( \*.hbm.xml ):

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

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

<hibernate-mapping>

<class name="com.javatpoint.mypackage.Employee" table="emp1000">

<id name="id">

<generator class="assigned"></generator>

</id>

<property name="firstName"></property>

<property name="lastName"></property>

</class>

</hibernate-mapping>

employee.hbm.xml

* Configuration file ( hibernate.cfg.xml ):

<?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="hbm2ddl.auto">update</property>

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

<property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>

<property name="connection.username">system</property>

<property name="connection.password">oracle</property>

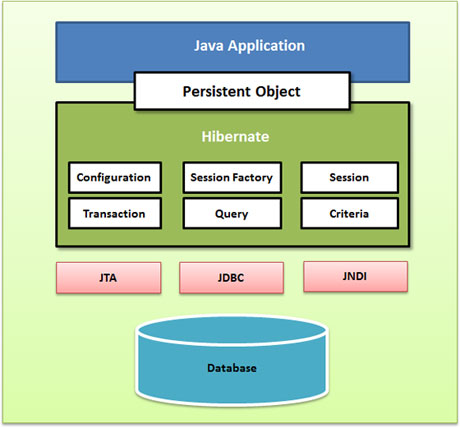
<property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>

<mapping resource="employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

* Session-factory: based on different servers (dialect.Oracle, dialect.MySql, dialect.SQLServer,jdbc:oracle, jdbc:mysql, ….



Hibernate use many objects.

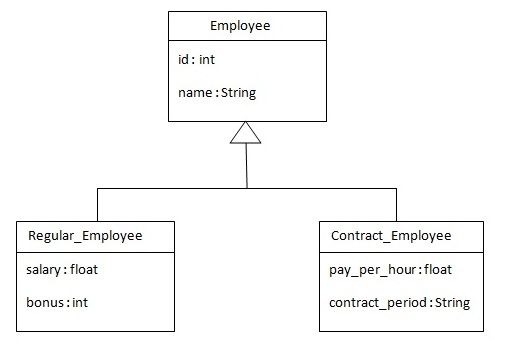
**Elements of Hibernate Architecture**

* Configuration: is the first Hibernate object which is created in any Hibernate app and usually created only once. Two keys components:
  + Database Connection: hibernate.cfg.xml and hibernate.properties
  + Class Mapping Setup: map Java class and database tables.
* SessionFactory: is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data, provides factory method to get the object of Session.
* Session: provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It hold a first-level cache (mandatory) of data. It provides methods to insert, update and delete the object, factory methods for Transaction, Query and Criteria.
* Transaction: “được ăn cả, ngã về không”, khi thực thi 1 khối lệnh, 1 câu lệnh xảy ra lỗi thì toàn bộ khối lệnh sẽ bị hủy, dữ liệu được giữ nguyên.

**Hibernate with Annotation**

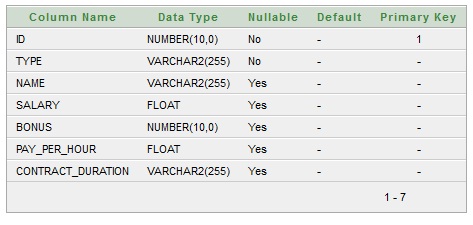
* Core advantage: don’t need to create mapping file (\*.hbm.xml). Hibernate annotations are used to provide the meta data.
* @Entity: marks this class as an entity.
* @Table(name=”\*\*\*”): to map the table name where data of this entity is to be persisted. If we don’t use @Table, hibernate will use the class name as the table name by default.
* @Id: marks primary key for entity bean (class).
* @GeneratedValue:
* @Column(name,length,nullable,unique): the details of the column for this property or field. If we don’t use @Column, property name will be used as the column name by default.

**Hibernate Inheritance Mapping**

****

Example structure

* Table Per Hierarchy: map the whole hierarchy by single table only (by xml or annotation). An extra column ( as <discriminator> column ) is created in the table to identify the class. (ex: TYPE column)



The single table structure

* Table Per Concrete Class: map table with table having no relations to each other by two ways. (xml or annotation).
  + By union-subclass element
  + By Self creating the table for each class.

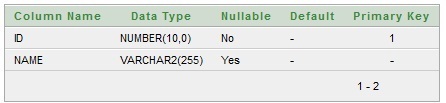


Table structure for Employee.class

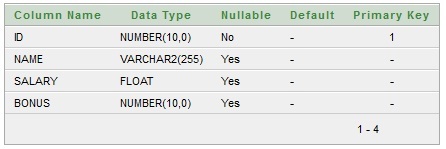


Table structure for Regular\_Employee class

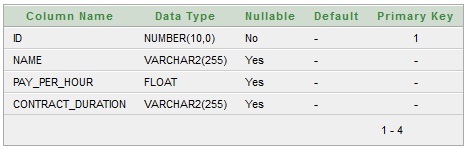


Table structure for Contract\_Employee class

* Table Per Subclass: subclass mapped tables are related to parent class mapped table by primary key and foreign key relationship.
  + The <joined-subclass> element: is used to map the child class with parent using the primary key and foreign key relation.
  + The <key column=”\*\*\*”> element of joined-subclass: is used to generate the foreign key in the subclass mapped table which will be associated with the primary key of parent class mapped table.

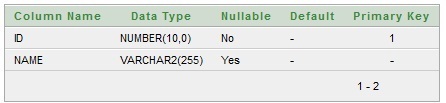


Table structure for Employee.class

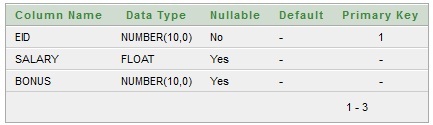


Table structure for Regular\_Employee.class

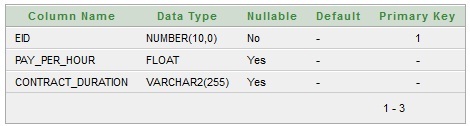


Table structure for Contract\_Employee.class

**Hibernate – Association Mappings**

* Many-to-One
* One-to-One
* One-to-Many
* Many-to-Many

**Hibernate – Collection Mapping**

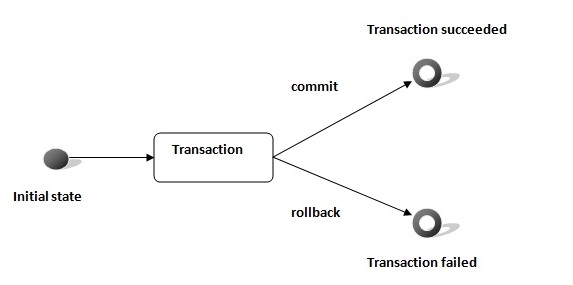
* Mapping List, Set, Map, SortedMap, SortedSet…
* By xml or annotation.

**Hibernate – Component Mapping**

* Mapping for a class having a reference to another class as a member variable.

**Hibernate – Transaction**

If one step fails, the whole transaction fails



* A transaction is associated with Session and instantiated by calling session.beginTransaction().
* Void begin(): start a new transaction.
* Void commit(): ends the unit of work unless we are in FlushMode.NEVER.
* Void rollback(): forces this transaction to rollback.
* Void setTimeout(int seconds): sets a transaction timeout for any transaction started.
* Boolean isAlive(): checks if the transaction is still alive.
* Void registerSynchronization(Synchronization s): registers a user synchronization callback for this transaction.
* Boolean wasCommited(): checks if the transaction is committed successfully.
* Boolean wasRolledBack(): checks if the transaction is rolledback successfully.

**Hibernate Query Language (HQL)**

* Is same as SQL but it doesn’t depends on the table of the database. Instead of table name, we use class name in HQL. Database independent query language.
* Advantages of HQL: database independent, supports polymorphic queries, easy to learn.
* Query Interface: by calling the createQuery() method Session interface.
  + Public int executeUpdate(): to execute the update or delete query.
  + Public List list(): returns result as a list.
  + Public Query setFirstResult(int rowno): specifies the row number from where record will be retrieved.
  + Public Query setMaxResutl(int rowno): specifies the no. of records to be retrieved from the relation (table).
  + Public Query setParameter(int position, Object value): it sets the value to the JDBC style query parameter.
  + Public Query setParameter(String name, Object value): it sets the value to a named query parameter.
* HQL with Aggregate functions: avg(), min(), max(), sum()…

**Hibernate Criteria Query Language (HCQL)**

* Is used to fetch the records based on the specific criteria. The Criteria interface provides methods to apply criteria such as retrieving all the records of table whose salary is greater than 50000, searching, etc.
* Criteria Interface
  + By calling the createCriteria() method of Session interface.
  + Syntax: public Criteria createCriteria(Class c)
  + Public Criteria add(Criterion c): is used to add restrictions.
  + Public Criteria addOrder(Order o): specifies ordering.
  + Public Criteria setFirstResult(int firstResult): specifies the first number of record to be retrieved.
  + Public Criteria setMaxResult(int totalResult): specifies the total number of records to be retrieved.
  + Public List list(): returns list containing object.
  + Public Criteria setProjection(Projection projection): specifies the projection.
* Restrictions class (RC)
  + RC.lt(String propertyName, Object value)
    - RC.gt(“salary”,1000); >> get the records whose salary greater than 1000.
  + RC.le(String propertyName, Object value)
  + etc
* Order class (OC)
  + OC.asc(String propertyName);
  + OC.desc(String propertyName);

**Hibernate Named Query**

* Is way to use any query by some meaningful name ( like alias names). By annotation or mapping file(xml).

**Hibernate Caching**

* First lever cache: enabled by default. Session object holds data. Data will not be available to entire app. An app can use many session object.
* Second lever cache: need to be enabled. SessionFactory object holds data. Data will be available to entire app. (some vendors: JBoss cache, OS Cacher, EH Cache,…)

**Hibernate and Spring Integration**

* Don’t need to create the hibernate.cfg.xml. We can provide all the information in the applicationContext.xml.
* Spring Framework provides HibernateTemplate class, so we don’t need to create Configuration, build SessionFactory, Session, beginning and committing transaction etc. It saves a lot of code.
* HibernateTemplate class:
  + Void persist(Object entity)
  + Void saveOrUpdate(Object entity)
  + List loadAll(Class entityClass)
  + etc