Hibernate Concepts

Get vs load :

Proxy and Lazy initialisation

get() involves database hit if object doesn't exists in Session Cache and returns a fully initialized object which may involve several [database](http://javarevisited.blogspot.sg/2011/11/database-transaction-tutorial-example.html) call

while load method can return proxy in place and only initialize the object or hit the database if any method other than getId() is called on persistent or entity object. This [lazy initialization](http://javarevisited.blogspot.sg/2011/03/10-interview-questions-on-singleton.html) can save couple of database round-trip which result in better performance

get() returns the object by fetching it from database or from [hibernate cache](http://www.journaldev.com/2969/hibernate-caching-first-level-cache) whereas load() just returns the reference of an object that might not actually exists, it loads the data from database or cache only when you access other properties of the object.

Get method never returns a proxy, it either returns null or fully initialized Object, while load() method may return proxy, which is the object with ID but without initializing other properties, which is lazily initialized. If you are just using returned object for creating relationship and only need Id then load() is the way to go

**Behavior when Object is not found in Session Cache**

Apart from performance this is another difference between get and load which is worth remembering. get method of Hibernate Session class returns null if object is not found in cache as well as on database while load() method [throws](http://javarevisited.blogspot.sg/2012/02/difference-between-throw-and-throws-in.html) ObjectNotFoundException if object is not found on cache as well as on database but never return null.

Automatic dirty Checking

Hibernate knows and tracks the Persistent object and its state change. Merely committing the transaction will cause the new data to be synchronized to the database with an update statement. Hibernate monitors all Persistent objects (i.e. the persistent context). At the end of a unit of work, it knows which objects have been modified. It then calls update statements on all updated objects. This process of monitoring and updating only objects that have changed is called automatic dirty checking.

Session session = sessionFactory.openSession();   
Transaction transaction = session.beginTransaction();   
BallPlayer p = (BallPlayer)session.get(BallPlayer.class, 1L);   
p.setNickname("Bambino");   
transaction.commit(); //new nickname is synch’ed and committed here   
session.close();

Output- Update Player set name=?, nickname=?, date\_of\_birth=?, city\_of\_birth=?, uniform\_number=? Where id=?

An object with modifications that have not yet been propagated to the database is considered *dirty*

Transactional Write behind

Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called *transactional write-behind.* With *transparent transaction-level write-behind*, Hibernate propagates

state changes to the database as late as possible but hides this detail from the application. By executing DML as late as possible (toward the end of the database transaction), Hibernate tries to keep lock-times in the database as short as possible.(DML usually creates locks in the database that are held until the transaction completes.)

**When objects associated with the persistence context are modified, the changes are not immediately propagated to the database**.  
What Hibernate does is instead **collect all such database operations associated with a transaction**, creating the minimum set of sql queries and executing them. This delayed execution of sql queries is known as **transactional write behind**. On calling a transaction.commit(), Hibernate flushes all the sql to the database. This flushing also happens :

1. **Before a query is executed.** Hibernate does not flush before every query. The flush is performed only if the sql statements in the session will affect the result of the query.
2. **When the application calls session.flush()**

The above behavior is hibernate FLASH mode as AUTO.Other possible values are – FlushMode.MANUAL

Session.setFlushMode(FlushMode.MANUAL);// In this case the flush happens only if an explicit call is made to session.flush()

**FlushMode.COMMIT** is used to ensure that the flush occurs when Transaction.commit() is called.

The other options include **FlushMode.ALWAYS** in which case the session is flushed before every query. This is an inefficient option because all objects in the persistence context need to be checked if they are dirty.

Chapter -9 Working with objects(Manning Java Peristence with Hibernate) book

* The Hibernate persistence manager, the Session, is responsible for managing object state
* Think of the persistence context as a cache that remembers all the modifications and state changes you made to objects in a particular unit of work
* Hibernate and Java Persistence consider all transient instances to be nontransactional; any modification of a transient instance isn’t known to a persistence context.
* Persistent instances are always associated with a *persistence context*. Hibernate caches them and can detect whether they have been modified by the application.
* An object is in the *removed* state if it has been scheduled for deletion at the end of a unit of work, but it’s still managed by the persistence context (session) until the unit of work completes.
* Persistence context can be considered as a cache of managed entity instances. In a Hibernate application, we say that one Session has one internal persistence context. In a Java Persistence application, an EntityManager has a persistence context.

The persistence context is useful for several reasons:

* Hibernate can do automatic dirty checking and transactional write-behind.
* Hibernate can use the persistence context as a first-level cache.
* Hibernate can guarantee a scope of Java object identity.
* Hibernate can extend the persistence context to span a whole conversation
* When a unit of work completes, state held in memory is propagated to the database by the execution of SQL INSERT, UPDATE, and DELETE statements (DML).

Two strategies are available to implement a conversation in a Hibernate or Java

* Persistence application: with detached objects or by extending a persistence context.

Objects are held in detached state

during user think-time, and any modification of these objects is made persistent

manually through reattachment or merging. This strategy is also called *session-perrequest-*

* *with-detached-objects*.

Hibernate Session save

**hibernate save()** can be used to save entity to database. We can invoke this method outside a transaction.  If we use this without transaction and we have cascading between entities, then only the primary entity gets saved **unless we flush the session**.

For example code refer the link -http://www.journaldev.com/3481/hibernate-session-merge-vs-update-save-saveorupdate-persist-example

* We should avoid save outside transaction boundary, otherwise mapped entities will not be saved causing data inconsistency. It’s very normal to forget flushing the session because it doesn’t throw any exception or warnings.
* Hibernate save method returns the generated id immediately, this is possible because primary object is saved as soon as save method is invoked.
* If there are other objects mapped from the primary object, they gets saved at the time of committing transaction or when we flush the session.
* For objects that are in persistent state, save updates the data through update query. Notice that it happens when transaction is committed. If there are no changes in the object, there wont be any query fired. If you will run above program multiple times, you will notice that update queries are not fired next time because there is no change in the column values.
* Hibernate save load entity object to persistent context, if you will update the object properties after the save call but before the transaction is committed, it will be saved into database.

Hibernate Persist

Similar to save(),difference is that we can use persist() method only within the boundary of a transaction, so it’s safe and takes care of any cascaded objects.

save() is guaranteed to assign and return an ID for the entity, whereas persist() is not.

**Hibernate saveOrUpdate**

**Hibernate saveOrUpdate** results into insert or update queries based on the provided data. If the data is present in the database, update query is executed.

We can use saveOrUpdate() without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed.

Hibernate saveOrUpdate adds the entity object to persistent context and track any further changes. Any further changes are saved at the time of committing transaction, like persist.

With transaction also, cascaded objects are saved or updated only at the time of tx.commit, whereas the primary object is inserted as soon as saveorupdate is done..Any further changes are tracked and propagated into DB once tx.commit is done.

### Hibernate update

**Hibernate update** should be used where we know that we are only updating the entity information. This operation adds the entity object to **persistent context** and further changes are tracked and saved when transaction is committed

Example:

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Employee emp = (Employee) session.load(Employee.class, new Long(101));

System.out.println("Employee object loaded. " + emp);

tx.commit();

// update example

emp.setName("Updated name");

emp.getAddress().setCity("Bangalore");

Transaction tx7 = session.beginTransaction();

session.update(emp);

emp.setName("Final updated name");

System.out.println("13. Before committing update transaction");

tx7.commit();

System.out.println("14. After committing update transaction");

// Close resources

sessionFactory.close();

Output:

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_0\_, employee0\_.emp\_name as emp\_name2\_1\_0\_, employee0\_.emp\_salary as emp\_sala3\_1\_0\_, address1\_.emp\_id as emp\_id1\_0\_1\_, address1\_.address\_line1 as address\_2\_0\_1\_, address1\_.city as city3\_0\_1\_, address1\_.zipcode as zipcode4\_0\_1\_ from EMPLOYEE employee0\_ left outer join ADDRESS address1\_ on employee0\_.emp\_id=address1\_.emp\_id where employee0\_.emp\_id=?

Employee object loaded. Id= 101, Name= Test Emp, Salary= 1000.0, {Address= AddressLine1= Test address1, City=Test City, Zipcode=12121}

13. Before committing update transaction

Hibernate: update EMPLOYEE set emp\_name=?, emp\_salary=? where emp\_id=?

Hibernate: update ADDRESS set address\_line1=?, city=?, zipcode=? where emp\_id=?

14. After committing update transaction

### Hibernate Merge

**Hibernate merge** can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked.

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

Employee emp = (Employee) session.load(Employee.class, new Long(101));

System.out.println("Employee object loaded. " + emp);

tx.commit();

//merge example - data already present in tables

emp.setSalary(25000);

Transaction tx8 = session.beginTransaction();

Employee emp4 = (Employee) session.merge(emp);

System.out.println(emp4 == emp); // returns false

emp.setName("Test");

emp4.setName("Kumar");

System.out.println("15. Before committing merge transaction");

tx8.commit();

System.out.println("16. After committing merge transaction");

// Close resources

sessionFactory.close();

}

}

First exection output:

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_0\_, employee0\_.emp\_name as emp\_name2\_1\_0\_, employee0\_.emp\_salary as emp\_sala3\_1\_0\_, address1\_.emp\_id as emp\_id1\_0\_1\_, address1\_.address\_line1 as address\_2\_0\_1\_, address1\_.city as city3\_0\_1\_, address1\_.zipcode as zipcode4\_0\_1\_ from EMPLOYEE employee0\_ left outer join ADDRESS address1\_ on employee0\_.emp\_id=address1\_.emp\_id where employee0\_.emp\_id=?

Employee object loaded. Id= 101, Name= Final updated name, Salary= 1000.0, {Address= AddressLine1= Test address1, City=Bangalore, Zipcode=12121}

false

15. Before committing merge transaction

Hibernate: update EMPLOYEE set emp\_name=?, emp\_salary=? where emp\_id=?

16. After committing merge transaction

In further execution, output produced is:

Hibernate: select employee0\_.emp\_id as emp\_id1\_1\_0\_, employee0\_.emp\_name as emp\_name2\_1\_0\_, employee0\_.emp\_salary as emp\_sala3\_1\_0\_, address1\_.emp\_id as emp\_id1\_0\_1\_, address1\_.address\_line1 as address\_2\_0\_1\_, address1\_.city as city3\_0\_1\_, address1\_.zipcode as zipcode4\_0\_1\_ from EMPLOYEE employee0\_ left outer join ADDRESS address1\_ on employee0\_.emp\_id=address1\_.emp\_id where employee0\_.emp\_id=?

Employee object loaded. Id= 101, **Name= Kumar, Salary= 25000.0**, {Address= AddressLine1= Test address1, City=Bangalore, Zipcode=12121}

false

15. Before committing merge transaction

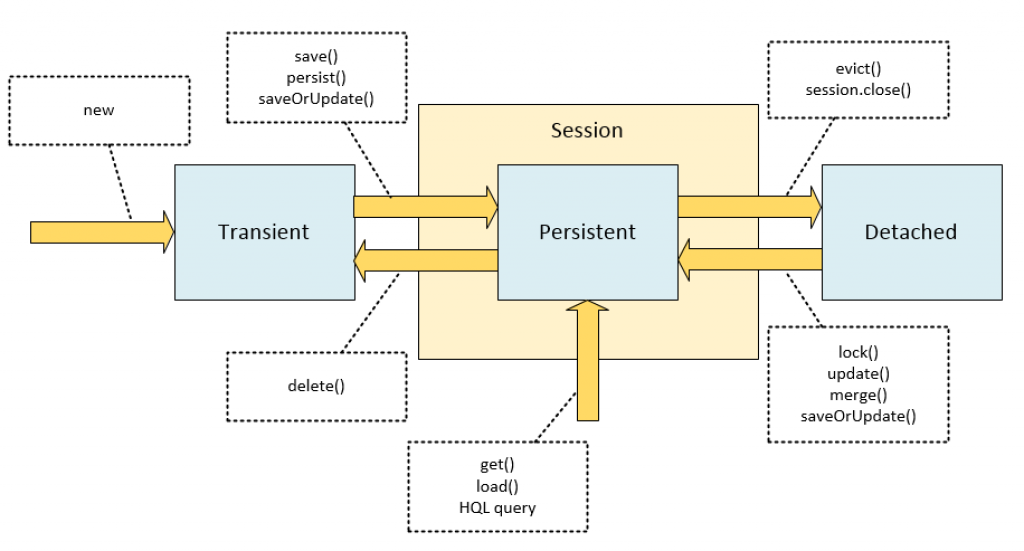
16. After committing merge transaction

Persistence context can be thought of as a container or a first-level cache for all the objects that you loaded or saved to a database during a session.

In Hibernate, the persistence context is represented by *[org.hibernate.Session](http://docs.jboss.org/hibernate/orm/5.2/javadocs/org/hibernate/Session.html)* instance. For JPA, it is the *[javax.persistence.EntityManager](http://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html)*. When we use Hibernate as a JPA provider and operate via EntityManager interface, the implementation of this interface basically wraps the underlying Session object. However, Hibernate Session provides a richer interface with more possibilities so sometimes it is useful  to work with Session directly.

States of entity instances

* *transient* — this instance is not, and never was, attached to a *Session*; this instance has no corresponding rows in the database; it’s usually just a new object that you have created to save to the database;
* *persistent* — this instance is associated with a unique *Session* object; upon flushing the *Session* to the database, this entity is guaranteed to have a corresponding consistent record in the database;
* *detached* — this instance was once attached to a *Session* (in a *persistent* state), but now it’s not; an instance enters this state if you evict it from the context, clear or close the Session, or put the instance through serialization/deserialization process.



save() is Hibernate-proprietary, whereas persist() is a standard JPA method.

The save method is an “original” Hibernate method that does not conform to the JPA specification.

Persist and merge conform to JPA specification . They are also portable in case you decide to switch to another persistence provider, but they may sometimes appear not so useful as the “original” Hibernate methods, save, update and saveOrUpdate.

when you try to save a *detached* instance:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | Person person = new Person();  person.setName("John");  Long id1 = (Long) session.save(person);    session.evict(person);  Long id2 = (Long) session.save(person); |

The *id2* variable will differ from *id1*. The call of save on a *detached* instance creates a new *persistent* instance and assigns it a new identifier, which results in a duplicate record in a database upon committing or flushing.

Hibernate Merge

The main intention of the merge method is to update a persistent entity instance with new field values from a detached entity instance.

* finds an entity instance by id taken from the passed object (either an existing entity instance from the persistence context is retrieved, or a new instance loaded from the database);
* copies fields from the passed object to this instance;
* returns newly updated instance.

we evict (detach) the saved entity from context, change the name field, and then merge the detached entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Person person = new Person();  person.setName("John");  session.save(person);    session.evict(person);  person.setName("Mary");    Person mergedPerson = (Person) session.merge(person); |

it is the mergedPerson object that was loaded into persistence context and updated, not the person object that you passed as an argument

Hibernate update

the *update* method is an “original” Hibernate method that was present long before the *merge* method was added. Its semantics differs in several key points:

* it acts upon passed object (its return type is *void*); the *update* method transitions the passed object from *detached* to *persistent* state;
* this method throws an exception if you pass it a *transient* entity.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Person person = new Person();  person.setName("John");  session.save(person);  session.evict(person);    person.setName("Mary");  session.update(person); |

Trying to call *update* on a *transient* instance will result in an exception. The following will not work:

|  |  |
| --- | --- |
| 1  2  3 | Person person = new Person();  person.setName("John");  session.update(person); // PersistenceException! |

Hibernate saveOrUpdate

The main difference of *saveOrUpdate* method is that it does not throw exception when applied to a *transient* instance; instead, it makes this *transient* instance *persistent*. The following code will persist a newly created instance of *Person*:

|  |  |
| --- | --- |
| 1  2  3 | Person person = new Person();  person.setName("John");  session.saveOrUpdate(person); |

It is a universal tool for making an object persistent regardless of its state wether it is transient or detached.

Update vs Merge

update() and merge() methods to transfer an object from detached state to persistent state.. When we call update(), if a session doesn’t contain same object (provided in update()) already in cache then update() method successfully executed and the object is converted detached state to persistent state. if already a session cache containing the same object then the update() method throws an exception called [NonUniqueObjectException](https://docs.jboss.org/hibernate/orm/3.5/api/org/hibernate/NonUniqueObjectException.html" \t "_blank).

public class HibernateUpdateExample {

    public static void main(String[] args) {

        Configuration configuration = new Configuration();

        configuration.configure("hibernate.cfg.xml");

        SessionFactory factory = configuration.buildSessionFactory();

        Session session = factory.openSession();

        Student student = (Student) session.get(Student.class, 111);

        session.close();

//          Here student object is in detached state

        student.setName("chandrashekhar");

//        reattaching to session

        Session session2 = factory.openSession();

        Student student2 = session2.get(Student.class, 111);

        Transaction tx = session2.beginTransaction();

        session2.update(student);

        tx.commit();

    }

}

Output :

Exception in thread "main" org.hibernate.NonUniqueObjectException:

A different object with the same identifier value was already associated with the session :

[com.onlinetutorialspoint.pojo.Student#111]

If we call merge() method, then it verifies whether the same object is existed in the cache or not. If the object is existed in the cache then changes are copied in to the cache. other wise it will load the values to cache. Hence it doesn’t throw any exception.

public class MergeExample {

    public static void main(String[] args) {

        Configuration configuration = new Configuration();

        configuration.configure("hibernate.cfg.xml");

        SessionFactory factory = configuration.buildSessionFactory();

        Session session = factory.openSession();

        Student student = (Student) session.get(Student.class, 111);

        session.close();

//          Here student object is in detached state

        student.setName("John");

//        reattaching to session

        Session session2 = factory.openSession();

        Student student2 = session2.get(Student.class, 111);

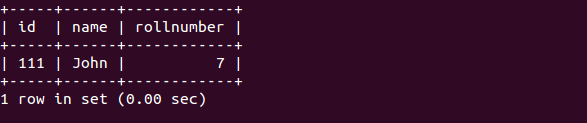
        Transaction tx = session2.beginTransaction();

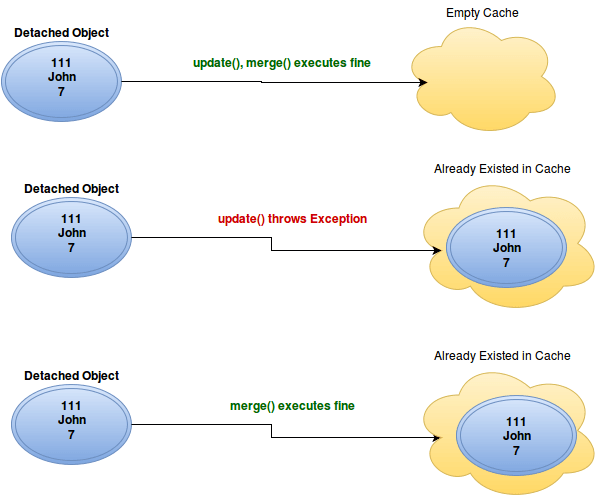
        session2.merge(student);

        tx.commit();

    }

}





When we call update() method, if the object already existed in cache update() method will throw exception where as merge() method copies the changes in to cache.

Lazy Loading

Say you have a parent and that parent has a collection of children. Hibernate now can "lazy-load" the children, which means that it does not actually load all the children when loading the parent. Instead, it loads them when requested to do so. You can either request this explicitly or, and this is far more common, hibernate will load them automatically when you try to access a child.

Lazy-loading can help improve the performance significantly since often you won't need the children and so they will not be loaded.

Also beware of the n+1-problem. Hibernate will not actually load all children when you access the collection. Instead, it will load each child individually. When iterating over the collection, this causes a query for every child. In order to avoid this, you can trick hibernate into loading all children simultaneously, e.g. by calling parent.getChildren().size().

The default behavior is to load ‘property values eagerly’ and to load ‘collections lazily’.

Also note that @OneToMany and @ManyToMany associations are defaulted to LAZY loading; and @OneToOne and @ManyToOne are defaulted to EAGER loading. This is important to remember to avoid any pitfall in future.

To understand N+1 problem, lets consider a scenario. Let’s say we have a collection of User objects mapped to t\_users table in database, and each user has collection or Role mapped to t\_roles table using a joining table t\_user\_roles. At the ORM level a User has many to many relationship with Role.

|  |
| --- |
| A user can have many roles. Roles are loaded Lazily. |

Now lets say we want to fetch all users from this table and print roles for each one. Very naive Object Relational implementation could be -

UserRepository with findAllBy method

public interface UserRepository extends CrudRepository<User, Long> {

List<User> findAllBy();

}

Equivalent SQL queries executed by ORM will be:

First Get All User (1)

Select \* from t\_users;

Then get roles for each user executed N times (where N is number of users)

Select \* from t\_user\_roles where userid = <userid>;

So we need one select for User and N additional selects for fetching roles for each user, where N is total number of users. This is a classic N+1 problem in ORM.

## N+1 Resolution

Hibernate & Spring Data JPA provide mechanism to solve the N+1 ORM issue.

**Spring Data JPA Approach**

If we are using Spring Data JPA, then we have two options to achieve this - using EntityGraph or using select query with fetch join.

public interface UserRepository extends CrudRepository<User, Long> {

List<User> findAllBy();

@Query("SELECT p FROM User p LEFT JOIN FETCH p.roles")

List<User> findWithoutNPlusOne();

@EntityGraph(attributePaths = {"roles"})

List<User> findAll();

}

|  |  |
| --- | --- |
|  | N+1 queries are issued at database level |
|  | using left join fetch, we resolve the N+1 problem |
|  | using attributePaths, Spring Data JPA avoids N+1 problem |

**2. Hibernate Approach**

If its pure Hibernate, then the following solutions will work.

Using HQL Query

"from User u join fetch u.roles roles roles"

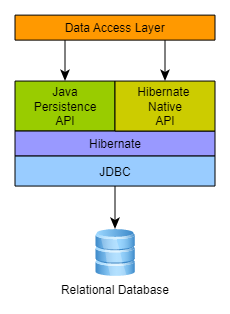
Using Hibernate Criteria API

Criteria criteria = session.createCriteria(User.class);

criteria.setFetchMode("roles", FetchMode.EAGER);

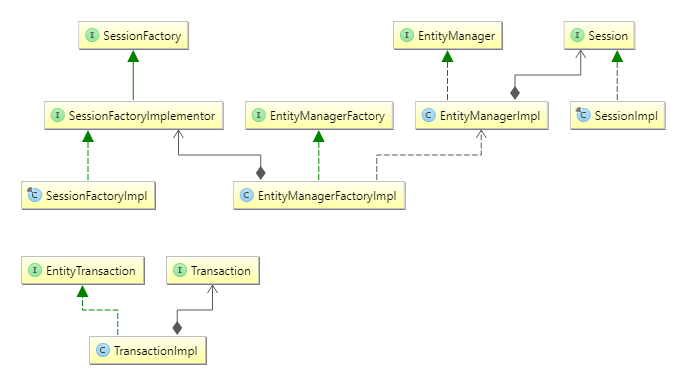
under the hood, all these approaches work similar and they issue a similar database query with left join fetch

### 1.1. Overview



Hibernate, as an ORM solution, effectively "sits between" the Java application data access layer and the Relational Database, as can be seen in the diagram above. The Java application makes use of the Hibernate APIs to load, store, query, etc its domain data. Here we will introduce the essential Hibernate APIs. This will be a brief introduction; we will discuss these contracts in detail later.

As a JPA provider, Hibernate implements the Java Persistence API specifications and the association between JPA interfaces and Hibernate specific implementations can be visualized in the following diagram:



SessionFactory (org.hibernate.SessionFactory)

A thread-safe (and immutable) representation of the mapping of the application domain model to a database. Acts as a factory for org.hibernate.Session instances. The EntityManagerFactory is the JPA equivalent of a SessionFactory and basically those two converge into the same SessionFactory implementation.

A SessionFactory is very expensive to create, so, for any given database, the application should have only one associated SessionFactory. The SessionFactory maintains services that Hibernate uses across all Session(s) such as second level caches, connection pools, transaction system integrations, etc.

Session (org.hibernate.Session)

A single-threaded, short-lived object conceptually modeling a "Unit of Work" [PoEAA](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#PoEAA). In JPA nomenclature, the Session is represented by an EntityManager.

Behind the scenes, the Hibernate Session wraps a JDBC java.sql.Connection and acts as a factory for org.hibernate.Transaction instances. It maintains a generally "repeatable read" persistence context (first level cache) of the application domain model.

Transaction (org.hibernate.Transaction)

A single-threaded, short-lived object used by the application to demarcate individual physical transaction boundaries. EntityTransaction is the JPA equivalent and both act as an abstraction API to isolate the application from the underlying transaction system in use (JDBC or JTA).

Multitenancy:

[Hibernate ORM 5.2.18.Final User Guide (jboss.org)](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#multitenacy)

### 19.1. What is multitenancy?

The term multitenancy, in general, is applied to software development to indicate an architecture in which a single running instance of an application simultaneously serves multiple clients (tenants).

The purpose is **to isolate the information each tenant needs from the shared database**.

Multitenancy Approach

1)Separate Database based

2)Separate Schema based

3)Partitioned/Discriminator based

Using Hibernate with multitenant data comes down to both an API and then integration piece(s)

The API is really just defined by passing the tenant identifier as part of opening any session.

*Example 612. Specifying tenant identifier from*SessionFactory

private void doInSession(String tenant, Consumer<Session> function) {

Session session = null;

Transaction txn = null;

try {

session = sessionFactory

.withOptions()

.tenantIdentifier( tenant )

.openSession();

txn = session.getTransaction();

txn.begin();

function.accept(session);

txn.commit();

} catch (Throwable e) {

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

throw e;

} finally {

if (session != null) {

session.close();

}

}

}

Additionally, when specifying the configuration, an org.hibernate.MultiTenancyStrategy should be named using the hibernate.multiTenancy setting. Hibernate will perform validations based on the type of strategy you specify. The strategy here correlates to the isolation approach discussed above.

#### **19.4.1. MultiTenantConnectionProvider**

When using either the DATABASE or SCHEMA approach, Hibernate needs to be able to obtain Connections in a tenant-specific manner.

That is the role of the MultiTenantConnectionProvider contract. Application developers will need to provide an implementation of this contract.

Most of its methods are extremely self-explanatory. The only ones which might not be are getAnyConnection and releaseAnyConnection. It is important to note also that these methods do not accept the tenant identifier. Hibernate uses these methods during startup to perform various configuration, mainly via the java.sql.DatabaseMetaData object.

#### **19.4.2. CurrentTenantIdentifierResolver**

org.hibernate.context.spi.CurrentTenantIdentifierResolver is a contract for Hibernate to be able to resolve what the application considers the current tenant identifier. The implementation to use is either passed directly to Configuration via its setCurrentTenantIdentifierResolver method. It can also be specified via the hibernate.tenant\_identifier\_resolver setting.

[java - What is HibernateTemplate class? - Stack Overflow](https://stackoverflow.com/questions/43801211/what-is-hibernatetemplate-class)

[Guide to JPA with Hibernate - Relationship Mapping (stackabuse.com)](https://stackabuse.com/a-guide-to-jpa-with-hibernate-relationship-mapping/)

JPA thought ahead and made One-to-Many relationships load lazily by default.

In our example, that would mean until we call on the Teacher#courses method, the courses are not being fetched from the database.

By contrast, Many-to-One relationships are eager by default, meaning the relationship is loaded at the same time the entity is.

We can change these characteristics by setting the fetch argument of both annotations:

@OneToMany(mappedBy = "teacher", fetch = FetchType.EAGER)

private List<Course> courses;

@ManyToOne(fetch = FetchType.LAZY)

private Teacher teacher;

That would inverse the way it worked initially. Courses would be loaded eagerly, as soon as we load a Teacher object. By contrast, the teacher wouldn't be loaded when we fetch courses if it's unneeded at the time.

**Optionality**

Considering the One-to-Many side - it is always optional, and we can't do anything about it. The Many-to-One side, on the other hand, offers us the option of making it mandatory.

let's remember that a relationship has an owning side - preferably the side which will hold the foreign key in the database.

1. Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it’s necessary to clear the cache at certain intervals to avoid memory issues.

## One small question, where the cache is stored?

In the Hibernate session is the hibernate’s first-level cache and SessionFactory is a second-level cache. So both (session/session-factory) are objects in a heap area. That means the cache is stored in the RAM only. And because of that, it gives faster access to data rather than databases.

[Hibernate Second-Level Cache | Baeldung](https://www.baeldung.com/hibernate-second-level-cache)

Hibernate second-level caching is designed to be unaware of the actual cache provider used. Hibernate only needs to be provided with an implementation of the org.hibernate.cache.spi.RegionFactory interface,

hibernate.cache.use\_second\_level\_cache=true hibernate.cache.region.factory\_class=org.hibernate.cache.ehcache.EhCacheRegionFactory

In order to **make an entity eligible for second-level caching**, we'll annotate it with the Hibernate specific *@org.hibernate.annotations.Cache* annotation, and specify a [cache concurrency strategy](https://www.baeldung.com/hibernate-second-level-cache#cacheConcurrencyStrategy).

@Entity

@Cacheable @org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)

**public** **class** **Foo** { @Id @GeneratedValue(strategy = GenerationType.AUTO) @Column(name = "ID") **private** **long** id; @Column(name = "NAME") **private** String name; // getters and setters }

### 21.2.2. Strategy: read only

If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used. This is the simplest and optimal performing strategy. It is even safe for use in a cluster.

### 21.2.3. Strategy: read/write

If the application needs to update data, a read-write cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required. **If the cache is used in a JTA environment, you must specify the property hibernate.transaction.manager\_lookup\_class and naming a strategy for obtaining the JTA TransactionManager. In other environments, you should ensure that the transaction is completed when Session.close() or Session.disconnect() is called**. If you want to use this strategy in a cluster, you should ensure that the underlying cache implementation supports locking. The built-in cache providers do not support locking.

### 21.2.4. Strategy: nonstrict read/write

If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate. If the cache is used in a JTA environment, you must specify hibernate.transaction.manager\_lookup\_class. In other environments, you should ensure that the transaction is completed when Session.close() or Session.disconnect() is called.

### 21.2.5. Strategy: transactional

The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.

* ***READ\_ONLY***: Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It's very simple and performative. It's suitable for static reference data that doesn't change.
* ***NONSTRICT\_READ\_WRITE***: Cache is updated after the transaction that changed the affected data has been committed. Thus, strong consistency isn't guaranteed, and there's a small time window in which stale data may be obtained from the cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency.
* ***READ\_WRITE***: This strategy guarantees strong consistency, which it achieves by using ‘soft' locks. When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from the database.
* ***TRANSACTIONAL***: Cache changes are done in distributed XA transactions. A change in a cached entity is either committed or rolled back in both the database and cache in the same XA transaction.
* Collections aren't cached by default, and we need to explicitly mark them as cacheable:
* @Entity
* @Cacheable
* @org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)
* **public** **class** **Foo** {
* ...
* @Cacheable
* @org.hibernate.annotations.Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)
* @OneToMany
* **private** Collection<Bar> bars;
* // getters and setters
* }

Hibernate Validtor:

Input validations can happen at different places in applications. Custom and possible duplicate code can be anywhere in the applications. Not to mention they are usually part of logic in the applications. Hibernate Validator is a reference implementation of Bean Validation (HTTP://BEANVALIDATION.ORG/). Bean Validation (added as part of Java EE 6) is a framework that defines a metadata model and API for JavaBeans validation. Constraints on JavaBeans can be expressed via annotations (the default metadata model) and can be extended through XML constraint mappings. Bean Validation 1.1 allows putting constraints to the parameters or return values on methods or constructors.

For Hibernate Validator, the current stable version is 5.2.2, at the time of writing. You can download it from HTTP://HIBERNATE.ORG/VALIDATOR/.

Composite primaty keys: 2 ways: either use @IDclass or use @Embedded

<https://www.baeldung.com/jpa-composite-primary-keys>

[JPA @Embedded And @Embeddable | Baeldung](https://www.baeldung.com/jpa-embedded-embeddable)

Spring Transaction Management:

The application server manages global transactions through the JTA, which is a cumbersome API to use (partly due to its exception model)

Typically you need an application server’s JTA capability only if your application needs to handle transactions across multiple resources, which is not a requirement for many applications

Standalone transaction managers such as [Atomikos Transactions](http://www.atomikos.com/" \t "_top) and [JOTM](http://jotm.objectweb.org/) are other options. Atomikos product – “Transaction Essentials” and “ExtremeTransactions”(professional mode)

TransactionsEssentials is our open source transaction management with JTA/XA and connection pooling for self-contained applications outside of the application server, ideal for your cloud.

ExtremeTransactions is our professional transaction management solution with JTA/XA, connection pooling and compensation-based TCC for self-contained applications outside of the application server, ideal for your cloud. Simple to install, implement and use and requires no additional services.

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

The related PlatformTransactionManager bean definition will then have a reference to the DataSource definition. It will look like this:

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

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

</bean>

 In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the SessionFactory.

<bean id="sessionFactory" class="org.springframework.orm.hibernate5.LocalSessionFactoryBean">

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

<property name="mappingResources">

<list>

<value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>

</list>

</property>

<property name="hibernateProperties">

<value>

hibernate.dialect=${hibernate.dialect}

</value>

</property>

</bean>

<bean id="txManager" class="org.springframework.orm.hibernate5.HibernateTransactionManager">

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

</bean>

* Unlike EJB CMT, which is tied to JTA, the Spring Framework’s declarative transaction management works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA, Hibernate or JDO by simply adjusting the configuration files.
* You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.

1. If Spring detects the @Transactional annotation on a bean, it creates a dynamic proxy of that bean.
2. The proxy has access to a transaction manager and will ask it to open and close transactions / connections.
3. The transaction manager itself will simply do what you did in the plain Java section: Manage a good, old JDBC connection.

Programmatic Spring Tx Management : [Programmatic Transaction Management in Spring | Baeldung](https://www.baeldung.com/spring-programmatic-transaction-management)

Use *TransactionTemplate:*

[*TransactionTemplate*](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/support/TransactionTemplate.html) provides a set of callback-based APIs to manage transactions manually. In order to use it, we should first initialize it with a PlatformTransactionManager.

**class** **ManualTransactionIntegrationTest** {

@Autowired **private** PlatformTransactionManager transactionManager;

**private** TransactionTemplate transactionTemplate;

@BeforeEach **void** **setUp**() {

transactionTemplate = **new** **TransactionTemplate**(transactionManager);

} // omitted

}

In addition to the TransactionTemplate, we can use an even lower-level API such as [PlatformTransactionManager](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/transaction/PlatformTransactionManager.html) to manage transactions manually. Quite interestingly, both @Transactional and TransactionTemplate use this API to manage their transactions internally.

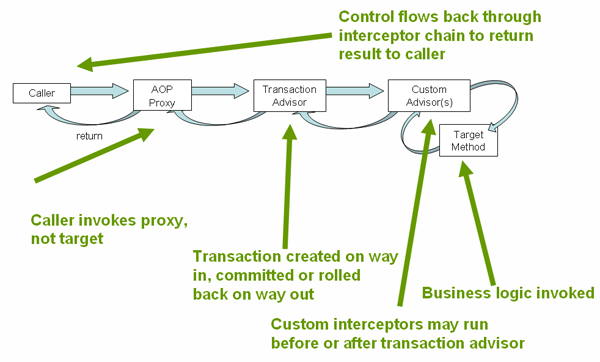
Isolation levels:

Table

Description automatically generated

[Spring transaction isolation level tutorial (byteslounge.com)](https://www.byteslounge.com/tutorials/spring-transaction-isolation-tutorial)

How declarative transactions work?



The above configuration is going to effect the creation of a transactional proxy around the object that is created from the 'fooService' bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked on the proxy, a transaction may be started, suspended, be marked as read-only, etc., depending on the transaction configuration associated with that method.

Tx:annotation-driven default settings: ([16. Transaction Management (spring.io)](https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/transaction.html))

| **XML Attribute** | **Annotation Attribute** | **Default** | **Description** |
| --- | --- | --- | --- |
| transaction-manager | N/A (See TransactionManagementConfigurer javadocs) | transactionManager | Name of transaction manager to use. Only required if the name of the transaction manager is not transactionManager, as in the example above. |
| mode | mode | proxy | The default mode "proxy" processes annotated beans to be proxied using Spring’s AOP framework (following proxy semantics, as discussed above, applying to method calls coming in through the proxy only). The alternative mode "aspectj" instead weaves the affected classes with Spring’s AspectJ transaction aspect, modifying the target class byte code to apply to any kind of method call. AspectJ weaving requires spring-aspects.jar in the classpath as well as load-time weaving (or compile-time weaving) enabled. (See [the section called “Spring configuration”](https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/aop.html#aop-aj-ltw-spring) for details on how to set up load-time weaving.) |
| proxy-target-class | proxyTargetClass | false | Applies to proxy mode only. Controls what type of transactional proxies are created for classes annotated with the @Transactional annotation. If the proxy-target-class attribute is set to true, then class-based proxies are created. If proxy-target-class is false or if the attribute is omitted, then standard JDK interface-based proxies are created. (See [Section 10.6, “Proxying mechanisms”](https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/aop.html#aop-proxying) for a detailed examination of the different proxy types.) |
| order | order | Ordered.LOWEST\_PRECEDENCE | Defines the order of the transaction advice that is applied to beans annotated with @Transactional. (For more information about the rules related to ordering of AOP advice, see [the section called “Advice ordering”](https://docs.spring.io/spring-framework/docs/4.2.x/spring-framework-reference/html/aop.html#aop-ataspectj-advice-ordering).) No specified ordering means that the AOP subsystem determines the order of the advice. |

By default @Transactional will rollback for all unchecked exceptions and will not rollback for checked exceptions. However this can be customized by changing rollbackfor property of @Transactional

@EnableTransactionManagement does the same thing for annotated configuration which <tx:annotation-driven> does for XML configuration.

[EnableTransactionManagement (Spring Framework 5.3.23 API)](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html)

@Configuration

@EnableTransactionManagement

public class AppConfig {

@Bean

public FooRepository fooRepository() {

// configure and return a class having @Transactional methods

return new JdbcFooRepository(dataSource());

}

@Bean

public DataSource dataSource() {

// configure and return the necessary JDBC DataSource

}

@Bean

public PlatformTransactionManager txManager() {

return new DataSourceTransactionManager(dataSource());

}

}

<beans>

<tx:annotation-driven/>

<bean id="fooRepository" class="com.foo.JdbcFooRepository">

<constructor-arg ref="dataSource"/>

</bean>

<bean id="dataSource" class="com.vendor.VendorDataSource"/>

<bean id="transactionManager" class="org.sfwk...DataSourceTransactionManager">

<constructor-arg ref="dataSource"/>

</bean>

</beans>

[**mode**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#mode--)

Indicate how transactional advice should be applied.(proxy or aspectj)

**The default is [AdviceMode.PROXY](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html" \l "PROXY).** Please note that proxy mode allows for interception of calls through the proxy only. Local calls within the same class cannot get intercepted that way; an [Transactional](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/Transactional.html) annotation on such a method within a local call will be ignored since Spring's interceptor does not even kick in for such a runtime scenario. For a more advanced mode of interception, consider switching this to [AdviceMode.ASPECTJ](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html" \l "ASPECTJ).

AspectJ mode

In this case, there will not be a proxy in the first place; instead, the target class will be weaved (that is, its byte code will be modified) in order to turn @Transactional into runtime behavior on any kind of method.

[**order**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#order--)

Indicate the ordering of the execution of the transaction advisor when multiple advices are applied at a specific joinpoint.

[**proxyTargetClass**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#proxyTargetClass--)

Indicate whether subclass-based (CGLIB) proxies are to be created (true) as opposed to standard Java interface-based proxies (false). The default is false. **Applicable only if**[**mode()**](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html#mode--)**is set to [AdviceMode.PROXY](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/context/annotation/AdviceMode.html" \l "PROXY)**.

**SPRING BOOT:** [What is Java Spring Boot? | IBM](https://www.ibm.com/cloud/learn/java-spring-boot)

Java Spring Framework (Spring Framework) is a popular, open source, enterprise-level framework for creating standalone, production-grade applications that run on the Java Virtual Machine (JVM).

Java Spring Boot (Spring Boot) is a tool that makes developing web application and microservices with Spring Framework faster and easier through three core capabilities:

1. Autoconfiguration
2. An opinionated approach to configuration
3. The ability to create standalone applications- Spring Boot helps developers create applications that just run. Specifically, it lets you create standalone applications that run on their own, without relying on an external web server, by embedding a web server such as Tomcat or Netty into your app during the initialization process.

**https://gainjavaknowledge.medium.com/how-spring-boot-application-works-internally-dd9bd3ecc487**

**High Level Flow Of Spring Boot And How run Method works :**  
================================

From the run method, the main application context is kicked off which in turn searches for the classes annotated with @Configuration, initializes all the declared beans in those configuration classes, and based upon the scope of those beans, stores those beans in JVM, specifically in a space inside JVM which is known as IOC container. After the creation of all the beans, automatically configures the dispatcher servlet and registers the default handler mappings, messageConverts, and all other basic things.

1. Basically, spring boot supports three embedded servers:- Tomcat (default), Jetty and Undertow.

[How Spring Boot Auto-Configuration Works - DZone Java](https://dzone.com/articles/how-springboot-autoconfiguration-magic-works)- (this is not so great link)

Spring Boot auto configuration examples:

* configuring a Data Source if Hibernate jar is on the class-path
* configuring a Dispatcher Servlet if Spring MVC jar is on the class-path.

Spring Boot looks at a) Frameworks available on the CLASSPATH b) Existing configuration for the application. Based on these, Spring Boot provides basic configuration needed to configure the application with these frameworks. This is called Auto Configuration.

As soon as we added in the Spring Boot Starter Web as a dependency in our project, Spring Boot Autoconfiguration sees that Spring MVC is on the classpath. It autoconfigures dispatcherServlet, a default error page and webjars.

If you add Spring Boot Data JPA Starter, you will see that Spring Boot Auto Configuration auto configures a datasource and an Entity Manager.

Spring boot starter:

First, let's look at developing the REST service; we can use libraries like Spring MVC, Tomcat and Jackson – a lot of dependencies for a single application.

Spring Boot starters can help to reduce the number of manually added dependencies just by adding one dependency. So instead of manually specifying the dependencies just add one starter as in the following example:

Notice that you don't need to specify the version number of an artifact. Spring Boot will figure out what version to use – all you need to specify is the version of spring-boot-starter-parent artifact. If later on you need to upgrade the Boot library and dependencies, just upgrade the Boot version in one place and it will take care of the rest.

out of the box we have automatic support for at least the following databases: H2, Derby and Hsqldb.

@SpringBootApplication internally contains below 3 annotations.

1. @Configuration to enable Java-based configuration
2. @ComponentScan to enable component scanning.
3. @EnableAutoConfiguration to enable Spring Boot's auto-configuration feature.

It also marks the class as a BootStrap class, which means you can runt it as a normal Java class, e.g. by running its JAR file

[Spring Boot](https://www.digitalocean.com/community/tutorials/spring-boot-tutorial) SpringApplication class is used to bootstrap and launch a Spring application from a Java main method. This class automatically creates the ApplicationContext from the classpath, scan the configuration classes and launch the application.

[Spring Boot - Customizing application startup (logicbig.com)](https://www.logicbig.com/tutorials/spring-framework/spring-boot/customizing-boot-startup.html)

we are using spring-boot-starter (core spring context). We are not even going to annotate the main class with @SpringBootApplication because we are not going to configure any beans in these examples:

package com.logicbig.example;

import org.springframework.boot.SpringApplication;

public class CustomizeBootExample {

public static void main (String[] args) {

SpringApplication app = new SpringApplication(CustomizeBootExample.class);

// customize start up here

app.run(args);

}

}

## Turning off banner

public static void main (String[] args) {

SpringApplication app = new SpringApplication(CustomizeBootExample.class);

app.setBannerMode(Banner.Mode.OFF);

app.run(args);

}

**CommandLineRunner Interface:** Command-line runners are a useful functionality to execute the various types of code that only have to be run once, right after application startup.

[Spring Boot CommandLineRunner Example Tutorial (javaguides.net)](https://www.javaguides.net/2020/02/spring-boot-commandlinerunner-example.html)

 two very popular interfaces in Spring Boot: **CommandLineRunner** and **ApplicationRunner**.

One common use case of these interfaces is to load some static data at application startup. Though, I have seen such usages mostly for test data setup only.

Both of them are functional interfaces with a run() method. **This run() method gets executed soon after the ApplicationContext is loaded and before SpringApplication#run method execution ends.**

**We have access to the application arguments as a raw String in the CommandLineRunner’s run() method.**

@Override

    public void run(**String[]** args) {

        LOG.info("Executing the command line runner, Application arguments: " + Arrays.toString(args));

    }

@Override

    public void run(**ApplicationArguments** args) {

        LOG.info("Executing SampleAppRunner");

    }

The ApplicationRunner provides access to the *[ApplicationArguments](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ApplicationArguments.html)*, not just the raw String arguments. Rather, **technically speaking, that’s the only difference between them.**

Multiple [CommandLineRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/CommandLineRunner.html" \o "interface in org.springframework.boot) beans can be defined within the same application context and can be ordered using the [Ordered](https://docs.spring.io/spring-framework/docs/5.3.23/javadoc-api/org/springframework/core/Ordered.html?is-external=true) interface or [@Order](https://docs.spring.io/spring-framework/docs/5.3.23/javadoc-api/org/springframework/core/annotation/Order.html?is-external=true) annotation.

If you need access to [ApplicationArguments](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ApplicationArguments.html" \o "interface in org.springframework.boot) instead of the raw String array consider using [ApplicationRunner](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/ApplicationRunner.html" \o "interface in org.springframework.boot).

Let’s say you have need to run a Scheduled batch Job, set some system environment properties, or need to perform some DB operation just before the Spring Boot run() method is finished, so in this kind of scenario ApplicationRunner Interface comes handy

[How to use ApplicationRunner in Spring Boot application? | Jhooq](https://jhooq.com/applicationrunner-spring-boot/)

* Spring Boot’s ‘Actuator’ dependency is used to monitor and manage the Spring web application.
* We can use it to monitor and manage the application with the help of HTTP endpoints or with the JMX.

To replace tomcat with jetty, you need to follow these steps.

1. Remove tomcat starter from spring-boot-starter-web
2. Add jetty starter
3. Adjust application.properties for any **server.tomcat.\*** entries( if available ).
4. <dependency>
5. <groupId>org.springframework.boot</groupId>
6. <artifactId>spring-boot-starter-web</artifactId>
7. <exclusions>
8. <exclusion>
9. <groupId>org.springframework.boot</groupId>
10. <artifactId>spring-boot-starter-tomcat</artifactId>
11. </exclusion>
12. </exclusions>
13. </dependency>
14. <dependency>
15. <groupId>org.springframework.boot</groupId>
16. <artifactId>spring-boot-starter-jetty</artifactId>
17. </dependency>

## What are some of the disadvantages of using Spring Boot?

Some of the common drawbacks of Spring Boot to keep in mind is its complexities. While working with Spring Boot, converting legacy to a full-fledged Spring Boot app could take up your time. Secondly, Spring Boot’s flexibility with large-scale projects isn’t suitable in the long run. As a developer, if you wish to choose Spring Boot to create monolithic apps, you will be disappointed. Finally, Spring Boot’s lack of control could multiply dependencies that could result in the execution of massive deployment data.

 Unlike the web applications, in **non-web applications** we have to implement CommandLineRunner interface. Hence, we have to provide implementation for its abstract run method. This method is a main method for our application as it the starting point for the app level logic. Moreover the the run method also passes all the command line arguments.

**Kubernetes**:Kubernetes is an open source orchestration system for Docker containers.

Kubernetes will execute your Docker containers and it will manage your Docker containers.

Although Kubernetes is an open source, but that is implemented by the Google.

Kubernetes is a tool which will manage your complete containerised

application.

It will manage the deployment, monitoring, execution and a lot of things.

At a high level Kubernetes workflow will look like this.

Here, the end user can submit the request to community cluster, either by the community CLI, or you

can submit the request or submit the job by the rest APIs or Kubernetes UI dashboard.

That event will be accepted by the community's master node, which have the multiple components and

the combination of these components called the control plane.

Within the control plane you have the **Kube controller, Kube API server, Kube scheduler and Kube key value store, which is called etcd.**

Then that master node can connect with the worker nodes.

You can have the multiple worker nodes connected with a single master or multiple master.

And on the worker note, we have **the Kube proxy, we have the Kubelet, we have the PODs and services.**

**Kube API server is** an entry point within the cluster and that is a single destination for all the rest APIs.

Whatever you are submitting to your Kubernetes cluster within the Kubernetes cluster, please make sure

all the requests accepted by the communities that is being done by the rest APIs.

So Kube API server will manage the complete inflow and outflow for the Kubernetes cluster.

Kube API server is the only interaction point with the Kubernetes.

**ETCD** is a key value Or you can say that is a distributed database for the Kubernetes. The complete state of your Kubernetes cluster is saved in the **ETCD** database.

It will provide the high availability of the data, which is related to the cluster.

And due to any case, if you lost your cluster, then you can recover your cluster with the **ETCD** database.

So on production machines, it is very necessary to externalise the ETCD database from Kubernetes cluster

because in any case, if my master node is going down, then I can recover the master node from the

ETCD database.

But if my ETCD is not externalized and that is present on the master node, then at the crash of Master,

the ETCD database will also crash and you will not be able to recover your Kubernetes cluster.

**Cube Scheduler** is basically used to schedule and regulate the jobs on Kubernetes cluster.

So scheduler, basically regulate the tasks on the slave node scheduler will get

the information regarding the resources which are present on the slave node, and then it will schedule the node on a suitable slave.

Controller in K8 will manage the bunch of jobs.

Whenever you have a bunch of jobs, controllers will execute these jobs or the controller utility as a single process.

**Worker Node** is a physical machine, or you can say that is a VM where the application or the containers are

actually executing and these applications and the containers are being managed by the master node. **Worker Node** contain the necessary service to manage the networking between the containers, the communication

with the master node, and assign the sources to the scheduled containers.

So ideally, the worker node can have all the capability to manage the containers and manage the resources,

the communication networking between the containers.

On each slave note or on each worker note.

You will get a utility called **Kubelet**

It is agent that execute on the worker nodes.

That agent will directly communicate with the Kube API server on the master node and that will manage

the state of your worker nodes. The execution of PODs or execution of the services on the worker node is responsibility of the **Kubelet**.

**Kubelet** will get the instructions from the cube API server that what is the service?

What is the configuration of that service and where it should need to execute on the worker node.

**POD** is a logical unit In K8.

You can have the single container or the multiple containers, but all the containers which are running

within a single Pod, they must have the similar kind of configuration which will share the storage, share the n/w. Another thing is that the containers within the POD can share the shared content and the same IP.

Within the Docker swarm.

IP is associated with the containers, but in K8 IP is associated with the POD, not with

the containers.

And the containers, which are a part of a single POD, they can communicate to each other via local

host.

If you have the cross configuration containers, then you need to define the separate ports for these

cross configuration containers.

Single POD can run on the multiple machines so you can execute a single POD on a multiple worker

nodes.

And a single machine can also execute the multiple PODs.

Cube proxy is a proxy which will deal with the individual host subnet.

Right.

And ensure that the service is available to the external parties.

Kube Proxy is a component which will provide the connectivity between the containers and the external world.

Helm in K8

 Helms, you can package your application and you can install your application.

You can also install some other plugins and other software, the third party software, which is required

for your Kubernetes cluster.

**REST**

REST is the acronym for REpresentational State Transfer. REST is an architectural style for developing applications that can be accessed over the network. REST architectural style was brought in light by Roy Fielding in his doctoral thesis in 2000. REST is a stateless client-server architecture where web services are resources and can be identified by their URIs. Client applications can use HTTP GET/POST methods to invoke Restful web services. REST doesn’t specify any specific protocol to use, but in almost all cases it’s used over HTTP/HTTPS. When compared to SOAP web services, these are lightweight and doesn’t follow any standard. We can use XML, JSON, text or any other type of data for request and response.

**Resource:**

Resource is the fundamental concept of Restful architecture. A resource is an object with a type, relationship with other resources and methods that operate on it. Resources are identified with their URI, HTTP methods they support and request/response data type and format of data.

SOAP stands for Simple Object Access Protocol. SOAP is an XML based industry standard protocol for designing and developing web services.

| SOAP is a standard protocol for creating web services.

| REST is an architectural style to create web services.

Restful web services supported HTTP methods are - GET, POST, PUT, DELETE and HEAD.

Service Oriented Architecture (SOA) is an architectural pattern where applications are designed in terms of services that can be accessed through communication protocol over network. SOA is a design pattern and doesn't go into implementation. Web Services can be thought of as Services in SOAP architecture and providing means to implement SOA pattern.

* REST is stateless, meaning the server does not store information about past communications with clients.

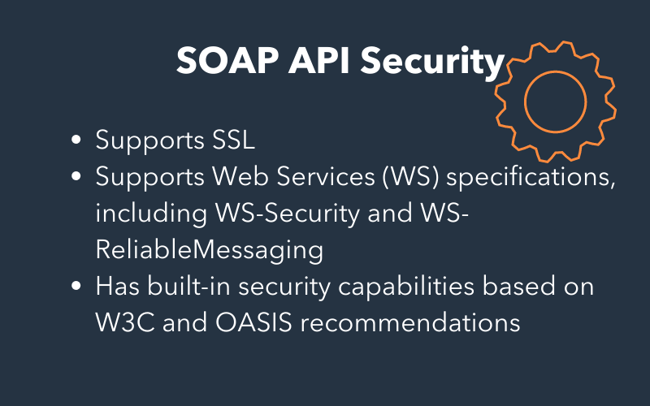
JAX-RS API:

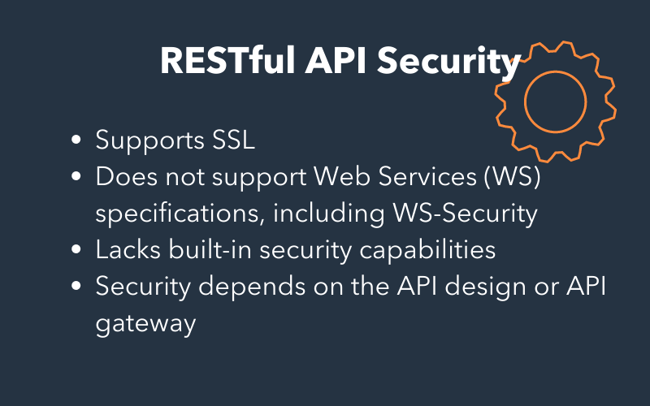
Java API for RESTful Web Services (JAX-RS) is the Java API for creating REST web services. JAX-RS uses annotations to simplify the development and deployment of web services. JAX-RS is part of JDK, so you don’t need to include anything to use it’s annotations.

There are two major implementations of JAX-RS API.

1. Jersey: Jersey is the reference implementation provided by Sun. For using Jersey as our JAX-RS implementation, all we need to configure its servlet in web.xml and add required dependencies. Note that JAX-RS API is part of JDK not Jersey, so we have to add its dependency jars in our application.

2. RESTEasy: RESTEasy is the JBoss project that provides JAX-RS implementation.





Java 8 features:

A Functional Interface is an interface that has exactly one abstract method.

The @FunctionalInterface annotation prevents abstract methods from being accidentally added to functional interfaces. It’s similar to a @Override annotation, and it’s recommended that you use it. java.lang. Runnable is a fantastic example of a functional interface since it has one abstract method, run ().

@FunctionalInterface

public interface FunctionalInterface\_one

{

public void firstInt\_method();

@Override

public String toString(); //Overridden from Object class

@Override

public boolean equals(Object obj); //Overridden from Object class

}

 Lambda Expressions implement functional interfaces by implementing the single abstract function provided in the functional interface.

importjava.util.ArrayList;

importjava.util.List;

public class Main {

public static void main(String[] args) {

List<String> subList = new ArrayList<String>();

subList.add("Carrot");

subList.add("Potato");

subList.add("Cauliflower");

subList.add("LadyFinger");

subList.add("Tomato");

System.out.println("------------Vegetable List--------------");

subList.forEach(sub -> System.out.println(sub));

}

[@FunctionalInterface](https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html)

public interface **Consumer<T>**

//abstract method

void accept([T](https://docs.oracle.com/javase/8/docs/api/java/util/function/Consumer.html) t)

}

public interface Iterable<T>{

Iterator<T> iterator();

**default** **void** forEach(Consumer<? **super** T> action) {

Objects.*requireNonNull*(action);

**for** (T t : **this**) {

action.accept(t);

}

}

Optional class:

the “java.util” package included an optional class. The public final class “Optional” is

You may use the Optional class to prevent the application from crashing and terminating unexpectedly. The Optional class has methods for checking the existence of a value for a given variable.used to handle NullPointerException in a Java program.

import java.util.Optional;

public class Main{

public static void main(String[] args) {

String[] str = new String[10];

Optional<String>checkNull =

Optional.ofNullable(str[5]);

if (checkNull.isPresent()) {

String word = str[5].toLowerCase();

System.out.print(str);

} else

System.out.println("string is null");

}

}

It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

public static void main(String[] args) {

Integer num = null;

Integer number = 18;

Integer orElse = Optional.ofNullable(num).orElse(12);

Integer anElse = Optional.ofNullable(number).orElse(12);

System.out.println("orElse: " + orElse);

System.out.println("anElse: " + anElse);

}

orElse: 12

anElse: 18

Optional is a ***value-based*** class, i.e their instances are :

* Final and immutable (though may contain references to mutable objects).
* Considered equal solely based on equals(), not based on reference equality(==).
* Do not have accessible constructors.

[Java 8 Stream Tutorial - GeeksforGeeks](https://www.geeksforgeeks.org/java-8-stream-tutorial/) : Stream API is used to process collections of objects

**Features of Java stream?**

* A stream is not a data structure instead it takes input from the Collections, Arrays**,** or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Streams:

w/o stream:

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

int count = 0;

for (String str : names) {

if (str.length() < 6)

count++;

}

System.out.println("There are "+count+" strings with length less than 6");

}

With Streams:

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

All these operations are happening parallelly which means we are able to parallelize the code with the help of streams. **Parallel execution of operations using stream is faster than sequential execution without using streams**.

## How to work with Stream in Java

As we have seen in the above example, the working of stream can be explained in three stages:  
1. Create a stream

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on on further intermediate operations. In the above example, the filter() operation is intermediate operation, there can be more than one intermediate operations.

3. Perform **terminal operation** on the final stream to get the result. In the above example, the count() operation is terminal operation.

// map() Operation

Stream strStream = Stream.of("Welcome", "To", "java", "blog");

Stream subStream2 = strStream.map(string -> {

if (string == "java")

return "Java-W3schools";

return string;

});

List welomeList = subStream2.collect(Collectors.toList());

System.out.println(welomeList);

Here, map function takes an argument type of Function. The function is a functional interface and has a method *apply()*. This method takes a value and returns another value.

*Here is the list of all Stream intermediate operations:*[***filter()***](https://www.javaprogramto.com/2019/12/java-8-stream-filter-example.html)[***map()***](https://www.javaprogramto.com/2019/12/java-8-stream-map-examples.html)[***flatMap()***](https://java8example.blogspot.com/2020/04/java-8-flatmap-examples-stream-flatmap.html)[***distinct()***](https://www.javaprogramto.com/2020/11/java-stream-distinct-by.html) *sorted()  
peek()  
limit()  
skip()*

### 5. Employees from SALES

List<Employee> salesEmployees = employees.stream()

.filter(employee -> employee.getDepartment().equals("SALES"))

.collect(Collectors.toList());

* 1. Stream<T> filter(Predicate<? super T> predicate);
  2. <R> Stream<R> map(Function<? super T,? extends R>mapper)
  3. Stream<T> sorted(Comparator<? super T> comparator);

Predicate is a functional interface with abstract method signature as:

boolean test(T t);

Function is a functional interface with abstract method signature as:

R apply(T t);

Comparator is a functional interface with abstract method:

int compare(T o1, T o2);

Map example:

List<Student> listOfStudents = createListOfStudents(); // Using map function to convert Stream<Student> to Stream<String> List<String> listOfStudentNames=listOfStudents.stream() .map(s -> s.getName()) .collect(Collectors.toList()); listOfStudentNames.forEach(System.out::println);

Sorted Example:

List<User> sortedList = userList.stream()

.sorted(Comparator.comparingInt(User::getAge).reversed())

.collect(Collectors.toList());

Another e.g

List<String> list = Arrays.asList("John", "Mark", "Robert", "Lucas", "Brandon");

List<String> sortedList = list.stream()

.sorted(Collections.reverseOrder())

.collect(Collectors.toList());

// creating a Stream of strings

        Stream<String> s = Stream.of("Geeks",

                                     "for",

                                     "GeeksforGeeks",

                                     "Geeks Classes");

        // using Collectors toSet() function

        Set<String> mySet = s.collect(Collectors.toSet());

OR

s.distinct().collect(Collectors.toList())

OR

s.distinct().collect(Collectors.toSet())

If you literally have .distinct followed immediately by .collect, it doesn't really add any benefit. Maybe if the .distinct implementation is more performant than the Set duplication check, you might get some benefit, but if you're collecting to a set you're going to end up with the same result anyway.

If, on the other hand, .distinct occurs before your .map operation, and that particular mapping is an expensive operation, you may get some gains there because you're processing less data overall.

an example of a terminal operation is[anyMatch](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#anyMatch-java.util.function.Predicate-). The[anyMatch](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#anyMatch-java.util.function.Predicate-) operation checks if any element in the input stream matches the condition specified. Consider the following code:

**public** **static** **void** main(String[] args) {

List<String> strList = Arrays.*asList*("Cat","Dog","Cow","Horse");

**boolean** anyMatch = strList.stream().anyMatch( str -> str.startsWith("C"));

System.***out***.println("Animal starting with C present: "+anyMatch);

}

Animal starting with C present: true

[Java 8 Streams Filter With Multiple Conditions Examples | JavaProgramTo.com](https://www.javaprogramto.com/2021/06/java-8-streams-filter-multiple-conditions.html)

[What is Diamond Problem in Java - Javatpoint](https://www.javatpoint.com/what-is-diamond-problem-in-java)

## **The Solution of Diamond Problem**

The solution to the diamond problem is **default methods** and **interfaces**. We can achieve multiple inheritance by using these two things.

1. **interface** DemoInterface1
2. {
3. **public** **default** **void** display()
4. {
5. System.out.println("the display() method of DemoInterface1 invoked");
6. }
7. }
8. **interface** DemoInterface2
9. {
10. **public** **default** **void** display()
11. {
12. System.out.println("the display() method of DemoInterface2 invoked");
13. }
14. }
15. **public** **class** DemoClass **implements** DemoInterface1, DemoInterface2
16. {
17. **public** **void** display()
18. {
19. DemoInterface1.**super**.display();
20. DemoInterface2.**super**.display();
21. }
22. **public** **static** **void** main(String args[])
23. {
24. DemoClass obj = **new** DemoClass();
25. obj.display();
26. }
27. }

# [**How does Spring Boot load changes in code without restarting the server**](https://stackoverflow.com/questions/45018873/how-does-spring-boot-load-changes-in-code-without-restarting-the-server)

Add spring-boot-devtools module to your project, which includes LiveReload server which can be used to trigger a browser refresh whenever a resource has been changed..

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

</dependencies>

Applications that use spring-boot-devtools will automatically restart whenever files on the classpath change.

The restart technology provided by Spring Boot works by using two classloaders. Classes that don’t change (for example, those from third-party jars) are loaded into a base classloader. Classes that you’re actively developing are loaded into a restart classloader. When the application is restarted, the restart classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than “cold starts” since the base classloader is already available and populated.

Executor framework simplifies the design of creating multithreaded application and manages thread life cycles.The programmer does not have to create or manage threads themselves, that’s the biggest advantage of executor framework.  
There are some important classes or interfaces for executor framework.

**Executor framework interfaces**

* **Executor:** It is used to submit a new task.
* **ExecutorService:** it is a subinterface of Executor that adds methods to manage lifecycle of threads. It is used to run the submitted tasks and methods to produce a Future to get a result from an asynchronous computation.
* **ScheduledExecutorService:** It is a subinterface of ExecutorService, to execute commands after a given delay.

// It creates a single thread ExecutorService

ExecutorService singleExecutorService = Executors.newSingleThreadExecutor();

//It creates a single thread ScheduledExecutorService

ScheduledExecutorService singleScheduledExecutorService = Executors.newSingleThreadScheduledExecutor();

//It creates an ExecutorService that use a pool of 5 threads

ExecutorService fixedExecutorService = Executors.newFixedThreadPool(5);

//It creates an ExecutorService that use a pool that creates threads on demand and kill them after 60 seconds if they are not used

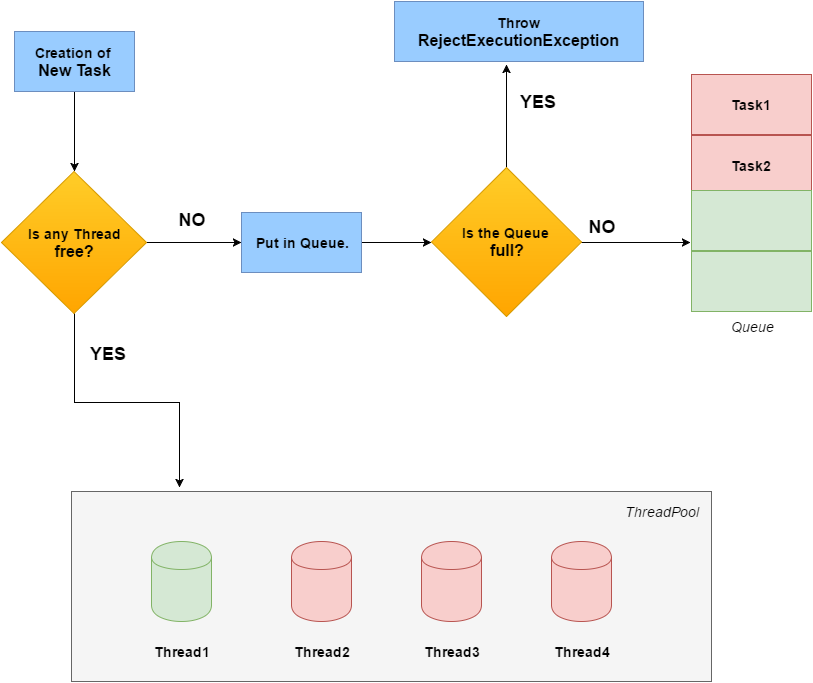
ExecutorService onDemandExecutorService = Executors.newCachedThreadPool();

//It creates a ScheduledExecutorService that use a pool of 5 threads

ScheduledExecutorService fixedScheduledExecutorService = Executors.newScheduledThreadPool(5);

[Executor Framework- Understanding the basics (Part 1) | by Anshul Jain | AndroidPub | Medium](https://medium.com/android-news/executor-framework-understanding-the-basics-43d575e72310)

The most commonly used implementations of these interfaces are [ThreadPoolExecutor](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html" \t "_blank) and [ScheduledThreadPoolExecutor](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html" \t "_blank)



Executors class has the method with below signature:

**public** **static** ExecutorService newFixedThreadPool(**int** nThreads) {

**return** **new** ThreadPoolExecutor(nThreads, nThreads,

0L, TimeUnit.***MILLISECONDS***,

**new** LinkedBlockingQueue<Runnable>());

}

If none of the executors provided by the above factory methods meet your needs, constructing instances of [java.util.concurrent.ThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html) or [java.util.concurrent.ScheduledThreadPoolExecutor](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledThreadPoolExecutor.html) will give you additional options.

**public** ThreadPoolExecutor(**int** corePoolSize,

**int** maximumPoolSize,

**long** keepAliveTime,

TimeUnit unit,

BlockingQueue<Runnable> workQueue, RejectedExecutionHandler handler)

By setting corePoolSize and maximumPoolSize the same, you create a fixed-size thread pool.

* **corePoolSize**: The minimum number of threads to keep in the pool.  
  **maximumPoolSize**: The maximum number of threads to keep in the pool.**keepAliveTime**: If current number of threads are greater than the minimum threads, then wait for this time to terminate the extra threads.**unit**: The [time unit](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/TimeUnit.html) for the previous argument.**workQueue**: The queue used for holding the tasks. If corePoolSize or more threads are running, the Executor always prefers queuing a request rather than adding a new thread.

**handler**: An instance of RejectionExecutionHandler, which handles the task which is rejected by the executor.

Case 1 Min =4 ,Max 14, current 2

Case 2 Min =4 ,Max 14, current 5

Case 2 Min =4 ,Max 14, current 14

Current< min

When a new task is submitted in method [execute(Runnable)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ThreadPoolExecutor.html#execute-java.lang.Runnable-), and fewer than corePoolSize threads are running, a new thread is created to handle the request, even if other worker threads are idle

Min < Current < Max

If there are more than corePoolSize but less than maximumPoolSize threads running, a new thread will be created **only if the queue is full.**

If the queue is not full, new thread is not created and task is queued.

Current =Max

* If a request cannot be queued(queue is full), the task will be rejected.
* Else put the task into queue.

Summary in my words:

If the min size is not reached, keep on creating threads for all incoming tasks. As soon as min size is reached, keep adding tasks to queue. Once the queue is full, then create further threads to process them.

**Example**

[Java Fixed Size Thread Pool Executor Example - HowToDoInJava](https://howtodoinjava.com/java/multi-threading/java-fixed-size-thread-pool-executor-example/)

**package** com.howtodoinjava.demo.multithreading;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.ThreadPoolExecutor;

**import** java.util.concurrent.TimeUnit;

**public** **class** FixedThreadPoolExecutorExample

{

**public** **static** **void** main(String[] args)

  {

    ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(4);

**for** (**int** i = 0; i < 10; i++)

    {

      Task task = **new** Task("Task " + i);

      System.out.println("A new task has been added : " + task.getName());

      executor.execute(task);

    }

    System.out.println("Maximum threads inside pool " + executor.getMaximumPoolSize());

    executor.shutdown();

  }

}

**class** Task **implements** Runnable

{

**private** String name;

**public** Task(String name)

  {

**this**.name = name;

  }

**public** String getName() {

**return** name;

  }

  @Override

**public** **void** run()

  {

**try**

    {

      Long duration = (**long**) (Math.random() \* 10);

      System.out.println("Doing a task during : " + name);

      TimeUnit.SECONDS.sleep(duration);

    }

**catch** (InterruptedException e)

    {

      e.printStackTrace();

    }

  }

}

## Shutting down an Executor

An executor can be shut down using [shutDown()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html" \l "shutdown%28%29" \t "_blank) function. When the executor is shut down, **it will no longer accept any new task** and submitting any task to it will throw a [RejectedExecutionException](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/RejectedExecutionException.html" \t "_blank). **But the tasks already executing on the threads and stored in the queue for execution will be executed.**But if we do not want to execute the tasks stored in queue, then we need to call [shutDownNow](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html" \l "shutdownNow%28%29" \t "_blank)().

What about the tasks which are already executing on the thread? Looks like calling any of the above two methods won’t cancel those tasks. But the executor sends an interrupt flag to all the currently executing threads. The runnable should check whether the thread is interrupted or not. And if the thread is interrupted, the runnable should decide what to do.

**public static** String getRandomString(**int** length) { **if**(Thread.currentThread().isInterrupted()){  
 // Either ignore this block or do some action  
 }  
}

Types of queues: **Size of the queue** : What should be size of the queue? Should the queue have a fixed size or should the queue be unbounded?

**Order of execution of the task** : What should be the order of execution of the tasks stored in the queue? Should a task which was submitted first be executed first or should the tasks be picked up based on their priority?

## [LinkedBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [PriorityBlockingQueue](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/PriorityBlockingQueue.html)

## This is a special type of queue in which the executor picks up a task from the queue based on its priority and not its order. While creating a task, we need to assign a priority to every task.

**Spring Boot @Async and @EnableAsync**

[Spring @Async to increase performance in 3 steps | SpringHow](https://springhow.com/spring-async/)

@EnableAsync

@SpringBootApplication

public class HelloWorldSpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(HelloWorldSpringBootApplication.class, args);

}

}

, @RestController

public class HelloWorldController {

private HelloService helloService;

public HelloWorldController(HelloService helloService) {

this.helloService = helloService;

}

@GetMapping("/hello")

public String hello() {

long start = System.currentTimeMillis();

helloService.processSomethingForLong();

long end = System.currentTimeMillis();

return "Hello World Took " + (end - start) + " milliseconds ! and the current Thread is : "+Thread.currentThread().getName();

}

## }

@Async

public void processSomethingForLong() {

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return "Hello World Took " + (end - start) + " milliseconds ! and the current Thread is : "+Thread.currentThread().getName();

}

**@Async with return types:**

@Async

public Future<String> longRunningProcessThatReturns() {

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return new AsyncResult<>("I take 10 seconds to return on a Thread named : " + Thread.currentThread().getName());

}

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

Future<String> stringFuture = helloService.longRunningProcessThatReturns();

while (!stringFuture.isDone()) {

Thread.sleep(1000);

}

String message = stringFuture.get();

The @EnableAsync annotation creates a handly **SimpleAsyncTaskExecutor**by default. Unfortunately,This implementation has no practical upper limit for its thread pool size. This means that The Spring boot application may crash if there were too many @Async methods running at the same time. To avoid this , we need to provide our own Executor.

As long as there is only one Bean of this type, you are good to go.

@Bean

Executor executor() {

return Executors.newFixedThreadPool(100);

}

We can have multiple executors in single application

@Bean("paymentExecutor")

Executor paymentExecutor() {

return Executors.newFixedThreadPool(100);

}

@Bean("deliveryExecutor")

Executor deliveryExecutor() {

return Executors.newFixedThreadPool(10);

}

@Async(value = "paymentExecutor")

public void doPayment(){

//perform payment

}

@Async(value = "deliveryExecutor")

public void doDeliveryArrangements(){

//perform delivery arrangements

}

**If you want to handle exceptions due to Async**

@Component

public class HelloAsyncConfigurer implements AsyncConfigurer {

@Override

public Executor getAsyncExecutor() {

return Executors.newFixedThreadPool(50);

}

@Override

public AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {

return (ex, method, params) -> {

System.out.println("Exception with message :" + ex.getMessage());

System.out.println("Method :" + method.toString());

System.out.println("Number of parameters :" + params.length);

};

}

}

**In w2 microservice we used below configuration**

@EnableAutoConfiguration (exclude = { DataSourceAutoConfiguration.**class** })

@SpringBootApplication

@EnableAsync

@EnableScheduling

**public** **class** W2ServiceApplication **extends** AsyncConfigurerSupport{

**public** Executor getAsyncExecutor() {

ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();

executor.setCorePoolSize(corePoolSize);

executor.setMaxPoolSize(maxPoolSize);

executor.setQueueCapacity(queueCapacity);

executor.setKeepAliveSeconds(1);

executor.setThreadNamePrefix("W2CostOfCoverage-");

executor.setTaskDecorator(**new** MDCCopyDecoratorWrapper());

executor.initialize();

**return** executor;

}

**Exception Handling in spring boot**

[Spring Boot - Exception Handling - GeeksforGeeks](https://www.geeksforgeeks.org/spring-boot-exception-handling/)

[Understanding Spring’s @ControllerAdvice | by Jovanny Cruz | Medium](https://medium.com/@jovannypcg/understanding-springs-controlleradvice-cd96a364033f)

@**ControllerAdvice** is very useful to handle exceptions when you have multiple Spring REST API controllers doing a lot of different work.

That means when writing any application you encounter exceptions and to handle them at each method level is tedious and not optimal. So in order to overcome that, spring has introduced the concept of @ControllerAdvice which will intercept all the controllers and look for the exceptions thrown. This is at a global level means you only have one @ControllerAdvice for each application and it will intercept the exceptions thrown by the controllers in that particular application context.

It would be great if they could create a centralized point for exceptions, so that whenever an exception is thrown by any controller, it would be caught and handled by that point instead of having handler methods in each controller class

Graphical user interface

Description automatically generated with low confidence

|  |
| --- |
| @ControllerAdvice |
|  | public class GlobalExceptionHandler { |
|  | /\*\* Provides handling for exceptions throughout this service. \*/ |
|  | @ExceptionHandler({ UserNotFoundException.class, ContentNotAllowedException.class }) |
|  | public final ResponseEntity<ApiError> handleException(Exception ex, WebRequest request) { |
|  | HttpHeaders headers = new HttpHeaders(); |
|  |  |
|  | if (ex instanceof UserNotFoundException) { |
|  | HttpStatus status = HttpStatus.NOT\_FOUND; |
|  | UserNotFoundException unfe = (UserNotFoundException) ex; |
|  |  |
|  | return handleUserNotFoundException(unfe, headers, status, request); |
|  | } else if (ex instanceof ContentNotAllowedException) { |
|  | HttpStatus status = HttpStatus.BAD\_REQUEST; |
|  | ContentNotAllowedException cnae = (ContentNotAllowedException) ex; |
|  |  |
|  | return handleContentNotAllowedException(cnae, headers, status, request); |
|  | } else { |
|  | HttpStatus status = HttpStatus.INTERNAL\_SERVER\_ERROR; |
|  | return handleExceptionInternal(ex, null, headers, status, request); |
|  | } |
|  | } |
|  |  |
|  | /\*\* Customize the response for UserNotFoundException. \*/ |
|  | protected ResponseEntity<ApiError> handleUserNotFoundException(UserNotFoundException ex, HttpHeaders headers, HttpStatus status, WebRequest request) { |
|  | List<String> errors = Collections.singletonList(ex.getMessage()); |
|  | return handleExceptionInternal(ex, new ApiError(errors), headers, status, request); |
|  | } |
|  |  |
|  | /\*\* Customize the response for ContentNotAllowedException. \*/ |
|  | protected ResponseEntity<ApiError> handleContentNotAllowedException(ContentNotAllowedException ex, HttpHeaders headers, HttpStatus status, WebRequest request) { |
|  | List<String> errorMessages = ex.getErrors() |
|  | .stream() |
|  | .map(contentError -> contentError.getObjectName() + " " + contentError.getDefaultMessage()) |
|  | .collect(Collectors.toList()); |
|  |  |
|  | return handleExceptionInternal(ex, new ApiError(errorMessages), headers, status, request); |
|  | } |
|  |  |
|  | /\*\* A single place to customize the response body of all Exception types. \*/ |
|  | protected ResponseEntity<ApiError> handleExceptionInternal(Exception ex, ApiError body, HttpHeaders headers, HttpStatus status, WebRequest request) { |
|  | if (HttpStatus.INTERNAL\_SERVER\_ERROR.equals(status)) { |
|  | request.setAttribute(WebUtils.ERROR\_EXCEPTION\_ATTRIBUTE, ex, WebRequest.SCOPE\_REQUEST); |
|  | } |
|  |  |
|  | return new ResponseEntity<>(body, headers, status); |
|  | } |
|  | } |

Another example of global exception handler

@ControllerAdvice

**public** **class** GlobalExceptionHandler {

    @ExceptionHandler(value

                      = NoSuchCustomerExistsException.**class**)

    @ResponseStatus(HttpStatus.BAD\_REQUEST)

**public** @ResponseBody ErrorResponse

    handleException(NoSuchCustomerExistsException ex)

    {

**return** **new** ErrorResponse(

            HttpStatus.NOT\_FOUND.value(), ex.getMessage());

    }

}

Handle exceptions in Java stream. Use Try(equivalent of Try in scala)

[How to handle exceptions properly within streams: the functional way | by Laurent Thiebaud | The Startup | Medium](https://medium.com/swlh/how-to-handle-exceptions-properly-within-steams-the-functional-way-f9da42c051a9)

|  |
| --- |
| public void answerRequestFunc(List<Integer> list) { |
|  | Stream<Try> result = list.stream().map(i -> { |
|  | try { |
|  | return new Success(i); |
|  | } catch (Exception excp) { |
|  | return new Failure(excp); |
|  | } |
|  | }); |
|  |  |
|  | result.forEach(r -> { |
|  | if (r.isFailure()) { |
|  | // report failures |
|  | System.out.print(r.getException()); |
|  | } else { |
|  | // report success |
|  | System.out.print(r.getValue())); |
|  | } |
|  | }); |
|  | } |

API Gateway:

With the Microservices pattern, a client may need data from multiple different microservices. If the client called each microservice directly, that could contribute to longer load times, since the client would have to make a network request for each microservice called. Moreover, having the client call each microservice directly ties the client to that microservice - if the internal implementations of the microservices change (for example, if two microservices are combined sometime in the future) or if the location (host and port) of a microservice changes, then every client that makes use of those microservices must be updated.

The intent of the API Gateway pattern is to alleviate some of these issues. In the API Gateway pattern, an additional entity (the API Gateway) is placed between the client and the microservices. The job of the API Gateway is to aggregate the calls to the microservices. Rather than the client calling each microservice individually, the client calls the API Gateway a single time. The API Gateway then calls each of the microservices that the client needs.

**MultiThreading**

*Each* ***process*** *has its own address space in memory, but* ***threads*** *share their address space*

Use of ThreadLocale variable:

For example, consider you are working on an eCommerce application. You have a requirement to generate a unique transaction id for each and every customer request this controller process and you need to pass this transaction id to the business methods in manager/DAO classes for logging purpose. One solution could be passing this transaction id as a parameter to all the business methods. But this is not a good solution as the code is redundant and unnecessary.

You can generate a transaction id in controller OR any pre-processor interceptor; and set this transaction id in the ThreadLocal. After this, whatever the methods, that this controller calls, they all can access this transaction id from the threadlocal. The transaction id will be unique to each thread and will be accessible from all over the thread’s execution path.

* Thread with the highest priority will get an execution chance prior to other threads. Suppose there are 3 threads t1, t2, and t3 with priorities 4, 6, and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
* If two threads have the same priority then we can’t expect which thread will execute first. It depends on the thread scheduler’s algorithm(Round-Robin, First Come First Serve, etc)
* If we are using thread priority for thread scheduling then we should always keep in mind that the underlying platform should provide support for scheduling based on thread priority.

If multiple threads are waiting to execute then thread execution is decided by “ThreadScheduler” which is a part of JVM hence its vendor dependent resulting in unexpected execution of output order.

* JVM terminates itself when all user threads finish their execution
* If JVM finds a running daemon thread, it terminates the thread and after that shutdown itself. JVM does not care whether the Daemon thread is running or not.
* It is an utmost low priority thread

Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");

Thread t3 = new MyThread("t3");

public class MyThread extends Thread {

public MyThread(String name) {

super(name);

}

@Override

public void run() {

System.out.println("MyThread - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("MyThread - END "+Thread.currentThread().getName());

}

private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

public class HeavyWorkRunnable implements Runnable {

@Override

public void run() {

System.out.println("Doing heavy processing - START "+Thread.currentThread().getName());

try {

Thread.sleep(1000);

//Get database connection, delete unused data from DB

doDBProcessing();

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("Doing heavy processing - END "+Thread.currentThread().getName());

}

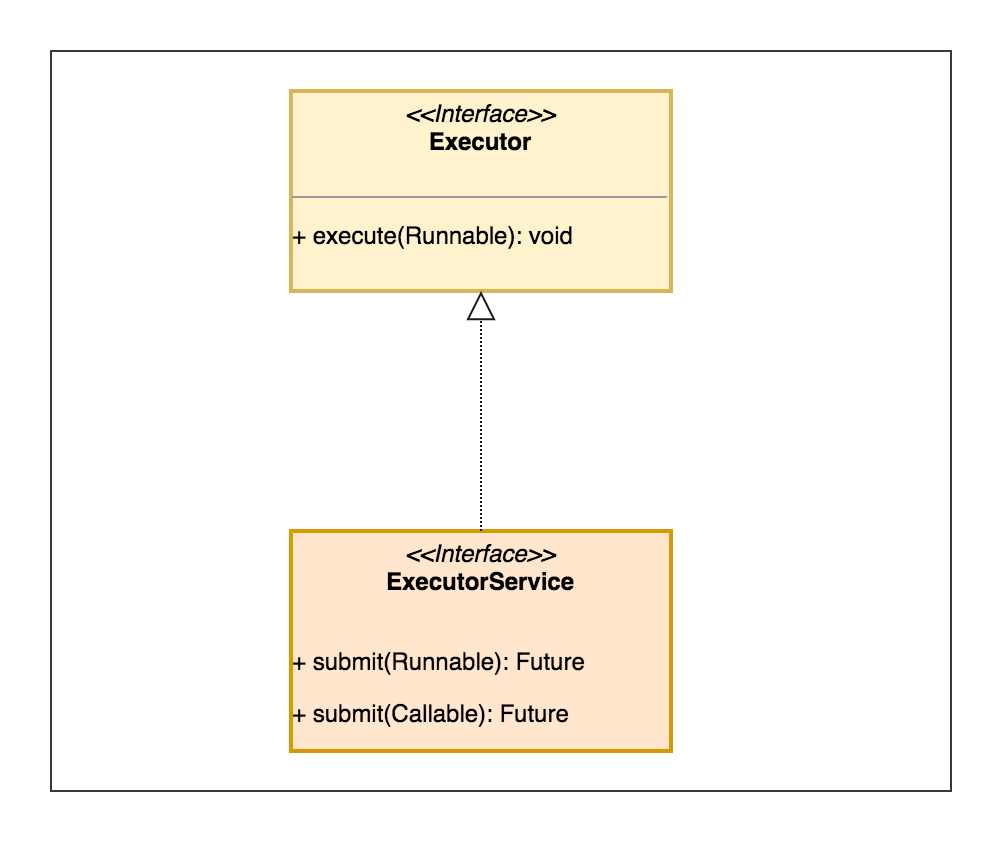
private void doDBProcessing() throws InterruptedException {

Thread.sleep(5000);

}

}

**Tip**: As you have noticed that thread doesn’t return any value but what if we want our thread to do some processing and then return the result to our client program, check our [**Java Callable Future**](https://www.digitalocean.com/community/tutorials/java-callable-future-example)



*Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable, however, does not return a result and cannot throw a checked exception.*

Callable<String> aCallable = () -> "dummy";

ExecutorService executorService = Executors.newSingleThreadExecutor();

final Future<String> future = executorService.submit(aCallable);

final String result = future.get();

// Class

// Implementing the Callable interface

**class** CallableMessage **implements** Callable<String>{

**public** String call() **throws** Exception{

**return** "Hello World!";

  }

}

**public** **class** CallableExample{

**static** ExecutorService executor = Executors.newFixedThreadPool(2);

**public** **static** **void** main(String[] args) **throws** Exception{

        CallableMessage task = **new** CallableMessage();

         Future<String> message = executor.submit(task);

         System.out.println(message.get().toString());

    }

}

**Java Thread join** method can be used to pause the current thread execution until unless the specified thread is dead. There are three overloaded join functions.

**public final void join()**: This java thread join method puts the current thread on wait until the thread on which it’s called is dead. If the thread is interrupted, it throws InterruptedException. **public final synchronized void join(long millis)**: This java thread join method is used to wait for the thread on which it’s called to be dead or wait for specified milliseconds. Since thread execution depends on OS implementation, it doesn’t guarantee that the current thread will wait only for given time. **public final synchronized void join(long millis, int nanos)**: This java thread join method is used to wait for thread to die for given milliseconds plus nanoseconds. Here is a simple example showing usage of **Thread join** methods. The goal of the program is to make sure main is the last thread to finish and third thread starts only when first one is dead.

|  |
| --- |
| **class** ThreadJoining **extends** Thread  {      @Override  **public** **void** run()      {  **for** (**int** i = 0; i < 2; i++)          {  **try**              {                  Thread.sleep(500);                  System.out.println("Current Thread: "                          + Thread.currentThread().getName());              }    **catch**(Exception ex)              {                  System.out.println("Exception has" +                                  " been caught" + ex);              }              System.out.println(i);          }      }  }    **class** GFG  {  **public** **static** **void** main (String[] args)      {            // creating two threads          ThreadJoining t1 = **new** ThreadJoining();          ThreadJoining t2 = **new** ThreadJoining();          ThreadJoining t3 = **new** ThreadJoining();            // thread t1 starts          t1.start();            // starts second thread after when          // first thread t1 has died.  **try**          {              System.out.println("Current Thread: "                    + Thread.currentThread().getName());              t1.join();          }    **catch**(Exception ex)          {              System.out.println("Exception has " +                                  "been caught" + ex);          }            // t2 starts          t2.start();            // starts t3 after when thread t2 has died.  **try**          {              System.out.println("Current Thread: "                   + Thread.currentThread().getName());              t2.join();          }    **catch**(Exception ex)          {              System.out.println("Exception has been" +                                      " caught" + ex);          }            t3.start();      }  } |

output:

Current Thread: main

Current Thread: Thread-0

0

Current Thread: Thread-0

1

Current Thread: main

Current Thread: Thread-1

0

Current Thread: Thread-1

1

Current Thread: Thread-2

0

Current Thread: Thread-2

1

wait():

When you call wait method on the object then it tell threads to give up the lock and go to sleep state unless and until some other thread enters in same monitor and calls notify or notifyAll methods on it.

This method tells the calling [**thread**](https://javagoal.com/multithreading-in-java/)(Current thread) to give up the lock and go to sleep until some other thread enters the same monitor and calls notify() or notifyAll().

notify():

When you call notify method on the object, it wakes one of thread waiting for that object. So if multiple threads are waiting for an object, it will wake of one of them. Now you must be wondering which one it will wake up. It actually depends on OS implementation.

### notifyAll() :

notifyAll will wake up all threads waiting on that object unlike notify which wakes up only one of them.Which one will wake up first depends on thread priority and OS implementation.

[wait, notify and notifyAll method in java with example - Java2Blog](https://java2blog.com/wait-notify-and-notifyall-method-in/)

[Producer-Consumer Problem With Example in Java | Baeldung](https://www.baeldung.com/java-producer-consumer-problem)

Java provides a BlockingQueue interface that is thread-safe. In other words, **multiple threads can add and remove from this queue without any concurrency issues**.

Its put() method blocks the calling thread if the queue is full. Similarly, if the queue is empty, its take() method blocks the calling thread.

In the produce() method, we can avoid explicit synchronization for our queue:

**private** **void** **produce**() {

**while** (true) {

**double** value = generateValue();

**try** {

blockingQueue.put(value);

} **catch** (InterruptedException e) {

**break**;

}

}

}

**private** **void** **consume**() { **while** (true) { Double value; **try** { value = blockingQueue.take(); } **catch** (InterruptedException e) { **break**; } // Consume value } }

**Cyclic Barrier:** [tutorials/CyclicBarrierDemo.java at master · eugenp/tutorials · GitHub](https://github.com/eugenp/tutorials/blob/master/core-java-modules/core-java-concurrency-advanced/src/main/java/com/baeldung/concurrent/cyclicbarrier/CyclicBarrierDemo.java)

The java.util.concurrent package contains several classes that help manage a set of threads that collaborate with each other. Some of these include:

* CyclicBarrier
* Phaser
* CountDownLatch
* Exchanger
* Semaphore
* SynchronousQueue

These classes offer out of the box functionality for common interaction patterns between threads.

If we have a set of threads that communicate with each other and resemble one of the common patterns, **we can simply reuse the appropriate library classes (also called Synchronizers) instead of trying to come up with a custom scheme using a set of locks and condition objects** and the synchronized keyword.

CyclicBarriers are used in programs in which we have a fixed number of threads that must wait for each other to reach a common point before continuing execution.

**CountDownLatch: Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0

[CountDownLatch in Java - GeeksforGeeks](https://www.geeksforgeeks.org/countdownlatch-in-java/)

For Difference refer to [Java CyclicBarrier vs CountDownLatch | Baeldung](https://www.baeldung.com/java-cyclicbarrier-countdownlatch)

**Cyclic Barrier vs CountdownLatch: -(Tasks vs. Threads)**

Let's take a deeper dive into some of the semantic differences between these two classes.

As stated in the definitions, CyclicBarrier allows a number of threads to wait on each other, whereas CountDownLatch allows one or more threads to wait for a number of tasks to complete.

In short, **CyclicBarrier maintains a count of threads** whereas **CountDownLatch maintains a count of tasks**.

In the following code, we define a CountDownLatch with a count of two. Next, we call countDown() twice from a single thread:

**CountDownLatch** countDownLatch = **new** **CountDownLatch**(2);

**Thread** t = **new** **Thread**(() -> {

countDownLatch.countDown();

countDownLatch.countDown();

});

t.start();

countDownLatch.await();

assertEquals(0, countDownLatch.getCount());Copy

Once the latch reaches zero, the call to await returns.

Note that in this case, **we were able to have the same thread decrease the count twice.**

**CyclicBarrier, though, is different on this point.**

Similar to the above example, we create a CyclicBarrier, again with a count of two and call await() on it, this time from the same thread:

**CyclicBarrier** cyclicBarrier = **new** **CyclicBarrier**(2);

**Thread** t = **new** **Thread**(() -> {

**try** {

cyclicBarrier.await();

cyclicBarrier.await();

} **catch** (InterruptedException | BrokenBarrierException e) {

// error handling

}

});

t.start();

assertEquals(1, cyclicBarrier.getNumberWaiting());

assertFalse(cyclicBarrier.isBroken());Copy

The first difference here is that the threads that are waiting are themselves the barrier.

Second, and more importantly, **the second await() is useless**.**A single thread can't count down a barrier twice.**

Indeed, because t must wait for another thread to call await() – to bring the count to two – t‘s second call to await() won't actually be invoked until the barrier is already broken!

In our test, **the barrier hasn't been crossed because we only have one thread waiting and not the two threads that would be required for the barrier to be tripped.** This is also evident from the cyclicBarrier.isBroken() method, which returns false.

The second most evident difference between these two classes is reusability. To elaborate, **when the barrier trips in CyclicBarrier, the count resets to its original value.** **CountDownLatch is different because the count never resets.**

**Cyclicbarrier says :” Call await method should be made by n-1 more threads, when await gets called by one thread”**

**Countdownlatch says :” call countdown method n times(by n threads or n times in single thread), when await gets called by main thread”**

**Semaphore:**

A semaphore controls access to a shared resource through the use of a counter. If the counter is greater than zero, then access is allowed. If it is zero, then access is denied. What the counter is counting are permits that allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

**Constructors in Semaphore class :** There are two constructors in Semaphore class.

Semaphore(int num)

Semaphore(int num, boolean how)

Here, num specifies the initial permit count. Thus, it specifies the number of threads that can access a shared resource at any one time. If it is one, then only one thread can access the resource at any one time. By default, all waiting threads are granted a permit in an undefined order. By setting how to true, you can ensure that waiting threads are granted a permit in the order in which they requested access.

* If the semaphore’s count is greater than zero, then the thread acquires a permit, which causes the semaphore’s count to be decremented.
* Otherwise, the thread will be blocked until a permit can be acquired.
* When the thread no longer needs an access to the shared resource, it releases the permit, which causes the semaphore’s count to be incremented.
* If there is another thread waiting for a permit, then that thread will acquire a permit at that time.

**Example :** [Semaphore in Java - GeeksforGeeks](https://www.geeksforgeeks.org/semaphore-in-java/)

In order to exclude the **instance variables** from the serialization process, we use the **transient** keyword. In order to indicate compiler to read the value of the variable from the main memory, we use the **volatile** keyword.

We can use the volatile keyword with **static** but cannot use the transient keyword with static.

**transient and static :** Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.

**transient and final :** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.

[Guide to the Volatile Keyword in Java | Baeldung](https://www.baeldung.com/java-volatile)

For multithreaded applications, we need to ensure a couple of rules for consistent behavior:

* Mutual Exclusion – only one thread executes a critical section at a time
* Visibility – changes made by one thread to the shared data are visible to other threads to maintain data consistency

### 3.3. volatile Memory Order

**To ensure that updates to variables propagate predictably to other threads, we should apply the volatile modifier to those variables:**

**public** **class** **TaskRunner** {

**private** **volatile** **static** **int** number;

**private** **volatile** **static** **boolean** ready;

// same as before

}

This way, we communicate with runtime and processor to not reorder any instruction involving the volatile variable. Also, processors understand that they should flush any updates to these variables right away.

synchronized methods and blocks provide both of the above properties at the cost of application performance.

volatile is quite a useful keyword because it **can help ensure the visibility aspect of the data change without providing mutual exclusion**. Thus, it's useful in the places where we're ok with multiple threads executing a block of code in parallel, but we need to ensure the visibility property.

@RequestMapping, @GetMapping and @PostMapping

* The @GetMapping annotation is a specialized version of [@RequestMapping](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.GET).
* The @PostMapping is a specialized version of @RequestMapping annotation that acts as a shortcut for @RequestMapping(method = RequestMethod.POST).

**@PathParam vs @PathVariable :** [@PathVariable](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVariable.html) and [@PathParam](https://docs.oracle.com/javaee/7/api/javax/ws/rs/PathParam.html) both are used for accessing parameters from [URI Template](https://en.wikipedia.org/wiki/URL_Template)

* As you mention @PathVariable is from spring and @PathParam is from [JAX-RS](https://en.wikipedia.org/wiki/Java_API_for_RESTful_Web_Services).
* @PathParam can use with REST only, where @PathVariable used in Spring so it works in MVC and REST.

***@PathParam***: it is used to inject the value of named URI path parameters that were defined in **@Path** expression.

@GET

@Path("/{make}/{model}/{year}")

@Produces("image/jpeg")

public Jpeg getPicture(@PathParam("make") String make, @PathParam("model") PathSegment car, @PathParam("year") String year) {

String carColor = car.getMatrixParameters().getFirst("color");

}

***@Pathvariable***: This annotation is used to handle template variables in the request URI mapping ,and used them as method parameters.

@GetMapping("/{id}")

public ResponseEntity<Patient> getByIdPatient(@PathVariable Integer id) {

Patient obj = service.getById(id);

return new ResponseEntity<Patient>(obj,HttpStatus.OK);

}

The @RequestParam is used to extract query parameters while @PathVariable is used to extract data right from the URI.

@GetMapping("/foos") @ResponseBody **public** String **getFooByIdUsingQueryParam**(@RequestParam String id) { **return** "ID: " + id; }

<http://localhost:8080/spring-mvc-basics/foos?id=abc>

----

ID: abc

@GetMapping("/foos/{id}") @ResponseBody **public** String **getFooById**(@PathVariable String id) { **return** "ID: " + id; }

<http://localhost:8080/spring-mvc-basics/foos/abc>

---- ID: abc

**Difference between ResponseEntity<> and @ResponseBody**

**HttpEntity** represents an HTTP **request** or **response** consists of **headers** and **body**.

**ResponseEntity** extends HttpEntity but also adds a Http status code. **ResponseEntity<>** is a generic class with a type parameter, you can specify what type of object to be serialized into the response body. ResponseEntity will give you some added flexibility in defining arbitrary HTTP response headers.

// i.e ResponseEntity = HttpEntity + StatusCode

public ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus statusCode)

@RequestMapping(value = "/message")

ResponseEntity<Message> get() {

Message message = new Message(penguinCounter.incrementAndGet() + " penguin!");

return new ResponseEntity<Message>(message, HttpStatus.OK);

}

**@ResponseBody** is an annotation, indicates that the return value of a method will be serialized into the body of the HTTP response.

@RequestMapping(value = "/message")

@ResponseBody

public Message get() {

return new Message(penguinCounter.incrementAndGet() + " penguin!");

}

DENSE\_RANK( ) OVER([ query\_partition\_clause ] order\_by\_clause)

SELECT

col,

DENSE\_RANK () OVER (

ORDER BY col )

col

FROM

dense\_rank\_demo;

with Result as

(

select dense\_rank () over (order by salary) as 'TopSalary',\* from table name

) select \* from Result where TopSalary between 1 and 5

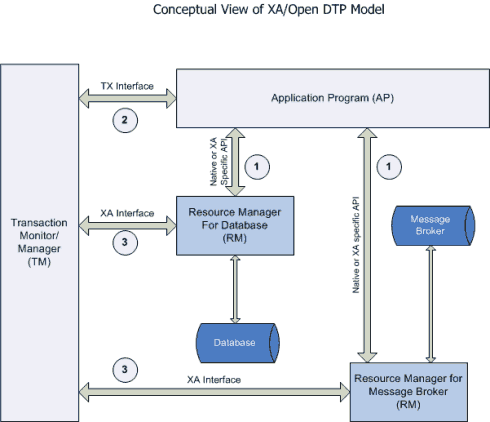
**how to achieve transaction when dealing with 2 different databases**

[XA transactions using Spring | InfoWorld](https://www.infoworld.com/article/2077714/xa-transactions-using-spring.html)

We can use distributed transactions (also called XA) in a standalone java application, without a JEE server, using the widely popular Spring framework and the open source JTA implementations of JBossTS, Atomikos and Bitronix.

Using distributed transactions, an application can accomplish tasks such as retrieving a message from a message queue and updating one or more databases in a single transactional unit adhering to the ACID (Atomicity, Consistency, Isolation and Durability) criteria.

1. The interface between the **application and the resource manager** allows an application to call a resource manager directly, using the resource manager's native API or native XA API depending on if the transaction needs to be managed by the transaction monitor or not.
2. The interface between the **application and the transaction monitor (TX interface),** lets the application call the transaction monitor for all transaction needs like starting a transaction, ending a transaction, rollback of a transaction etc.
3. The interface between the **transaction monitor and the resource manager** is the XA interface. This is the interface, which facilitates the two-phase commit protocol to achieve distributed transactions under one global transaction.



In order to be involved in an XA transaction, the XA Resource must make itself known to the Transaction Manager. This process is called enlistment. Once an XA Resource is enlisted, the Transaction Manager ensures that the XA Resource takes part in a transaction and makes the appropriate method calls on the XA Resource during the lifetime of the transaction. For an XA transaction to complete, all the Resource Managers participate in a two-phase commit (2pc). A commit in an XA transaction is called a two-phase commit because there are two passes made in the committing process. In the first pass, the Transaction Manager asks each of the Resource Managers (through the enlisted XA Resource) whether they will encounter any problems committing the transaction. If any Resource Manager objects to committing the transaction, then all work done by any party on any resource involved in the XA transaction must all be rolled back. The Transaction Manager calls the rollback() method on each of the enlisted XA Resources. However, if no resource Managers object to committing, then the second pass involves the Transaction Manager actually calling commit() on each of the enlisted XA Resources. This process guarantees the ACID (atomicity, consistency, isolation, and durability) properties of a transaction that can span multiple resources.

What is scoped bean injection problem in spring?

<https://justamonad.com/injecting-prototype-bean-in-singleton-bean/>

The default scope of a bean in Spring is singleton. It means that Spring will create one instance in its lifetime. Any request to bean with a matching id or ids results in the same instance.

The prototype scope means any request to bean with a matching id or ids results in creating a new bean.

Why would we really run into the problem when injecting a prototype bean into a singleton bean class? This is because the Singleton bean is initialized only once, at the start of Application context. Once the singleton bean is created, including its dependencies, we cannot to get any other beans into the class using injection. Problem arises when the singleton bean depends on the prototype bean.

**public** **class** **SingletonBean** {

// .. @Autowired **private** PrototypeBean prototypeBean;

**public** **SingletonBean**() {

logger.info("Singleton instance created");

}

**public** PrototypeBean **getPrototypeBean**() { logger.info(String.valueOf(LocalTime.now()));

**return** prototypeBean;

}

}

**public** **static** **void** **main**(String[] args) **throws** InterruptedException

{

**AnnotationConfigApplicationContext** context = **new** **AnnotationConfigApplicationContext**(AppConfig.class);

**SingletonBean** firstSingleton = context.getBean(SingletonBean.class); **PrototypeBean** firstPrototype = firstSingleton.getPrototypeBean(); // get singleton bean instance one more time

**SingletonBean** secondSingleton = context.getBean(SingletonBean.class); **PrototypeBean** secondPrototype = secondSingleton.getPrototypeBean();

isTrue(firstPrototype.equals(secondPrototype), "The same instance should be returned"); }

**Both beans were initialized only once,** at the startup of the application context.

We solve the scoped bean injection problem using the lookup method injection, ApplicationContext injection, and ObjectFactory and Provider interfaces to inject new instances for each invocation to the beans getter method.

### ****Using javax.inject.Provider<T>****

This is the simplest way to inject prototype beans in a singleton bean class. Provider<T> interface is part of the JSR-330 Spec.

The definition of this interface is for any call to the get() method of this interface, it will return an instance of T.

We need no changes in PrototypeBean class. SingletonBean class needs to change a little.

public class SingletonBean {

private final Provider<PrototypeBean> prototypeBean;

@Inject

public SingletonBean(Provider<PrototypeBean> prototypeBean) {

this.prototypeBean = prototypeBean;

}

public void singletonBeanMethod() {

System.out.println("Called singletonBeanMethod.");

System.out.println("SingletonBean hashCode is "

+ this.hashCode());

prototypeBean.get().prototypeBeanMethod();

}

 If you are using Spring annotations all over the place, then you should use ObjectFactory.

### ****Using Spring ObjectFactory<T>****

@Named

public class SingletonBean {

private final ObjectFactory<PrototypeBean> prototypeBean;

@Inject

public SingletonBean( ObjectFactory<PrototypeBean> prototypeBean) {

this.prototypeBean = prototypeBean;

}

public void singletonBeanMethod() {

System.out.println("Called singletonBeanMethod."); System.out.println("SingletonBean hashCode is " + this.hashCode()); prototypeBean.getObject().prototypeBeanMethod();

}

### ****Using proxyMode = ScopedProxyMode.TARGET\_CLASS****

One way to inject prototype bean into a singleton bean is to create a proxy of the prototype bean class. The proxy will be created just once and will be returned whenever the getBean() method of BeanFactory interface is called. When you invoke a method on this proxy, based on the scope of this bean it will either create a new instance or reuse the existing one. As the scope is set to prototype it will create a new instance every time.

@Named

@Scope(value = ConfigurableBeanFactory.SCOPE\_PROTOTYPE, proxyMode = ScopedProxyMode.TARGET\_CLASS)

public class PrototypeBean {

public void prototypeBeanMethod() {

System.out.println("Called prototypeBeanMethod.");

System.out.println("PrototypeBean hashCode is "

+ this.hashCode());

}

Singleton class can now directly inject PrototypeBean.

}

@Named

public class SingletonBean {

private final PrototypeBean prototypeBean;

@Inject

public SingletonBean(PrototypeBean prototypeBean) {

this.prototypeBean = prototypeBean;

}

public void singletonBeanMethod() {

System.out.println("Called singletonBeanMethod.");

System.out.println("SingletonBean hashCode is "

+ this.hashCode());

prototypeBean.prototypeBeanMethod();

}

}

**Injecting ApplicationContext:**

**To achieve this, either use the @Autowire annotation or implement the ApplicationContextAware interface:**

**public** **class** **SingletonAppContextBean** **implements** **ApplicationContextAware** {

**private** ApplicationContext applicationContext;

**public** PrototypeBean **getPrototypeBean**() {

**return** applicationContext.getBean(PrototypeBean.class);

}

@Override

**public** **void** **setApplicationContext**(ApplicationContext applicationContext)

**throws** BeansException {

this.applicationContext = applicationContext; } }

Every time the getPrototypeBean() method is called, a new instance of PrototypeBean will be returned from the ApplicationContext.

**However, this approach has serious disadvantages.** It contradicts the principle of inversion of control, as we request the dependencies from the container directly.

Also, we fetch the prototype bean from the applicationContext within the SingletonAppcontextBean class. **This means** **coupling the code to the Spring Framework**.

Another way to solve the problem is method injection with the **@Lookup annotation**:

@Component **public** **class** **SingletonLookupBean** {

@Lookup

**public** PrototypeBean **getPrototypeBean**() {

**return** null;

}

}

Spring will override the getPrototypeBean() method annotated with @Lookup. It then registers the bean into the application context. Whenever we request the getPrototypeBean() method, it returns a new PrototypeBean instance.

**It will use CGLIB to generate the bytecode** responsible for fetching the PrototypeBean from the application context.

**A method annotated with @Lookup tells Spring to return an instance of the method's return type when we invoke it.**

Essentially, Spring will override our annotated method and use our method's return type and parameters as arguments to BeanFactory#getBean.

## **8. Create a Bean at Runtime Using**java.util.Function

See [Injecting Prototype Beans into a Singleton Instance in Spring | Baeldung](https://www.baeldung.com/spring-inject-prototype-bean-into-singleton)

Next, we'll inject a bean factory into our singleton bean by making use of the java.util.Function interface:

**public** **class** **SingletonFunctionBean** { @Autowired **private** Function<String, PrototypeBean> beanFactory; **public** PrototypeBean **getPrototypeInstance**(String name) { **PrototypeBean** bean = beanFactory.apply(name); **return** bean; } }

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

**@autowrired vs @Inject**

### @Autowired(required=false)

By default the dependency injection for @Autowired must be fulfilled because the value of required attribute is true by default. We can change this behavior by using @Autowired(required=false). In this case if bean is not found for dependency injection, it will not through error.

@Service

public class EmployeeService {

private Employee employee;

public Employee getEmployee() {

return employee;

}

@Autowired(required=false)

public void setEmployee(Employee employee) {

this.employee = employee;

}

}

best way to save / modify 1 lakh records in hibernate

I would not use Session for bulk operations. StatelessSesssion is meant to do bulk operations like this as it doesn't maintain 1st level cache: Change "sessionFactory.openSession()" to "sessionFactory.openStatelessSession()"

StatelessSession session = sessionFactory.openStatelessSession();

Transaction tx = session.beginTransaction();

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

Item item = new Item(...);

session.insert(item);

}

tx.commit();

session.close();

In particular, a stateless session does not implement a first-level cache nor interact with any second-level or query cache. It does not implement transactional write-behind or automatic dirty checking. Operations performed using a stateless session never cascade to associated instances. Collections are ignored by a stateless session. Operations performed via a stateless session bypass Hibernate's event model and interceptors. Due to the lack of a first-level cache, Stateless sessions are vulnerable to data aliasing effects.

**Singleton Pattern:**

**class** Singleton

{

**private** **static** **volatile** Singleton obj  = **null**;

**private** Singleton() {}

**public** **static** Singleton getInstance()

    {

**if** (obj == **null**)

        {

            // To make thread safe

**synchronized** (Singleton.**class**)

            {

                // check again as multiple threads

                // can reach above step

**if** (obj==**null**)

                    obj = **new** Singleton();

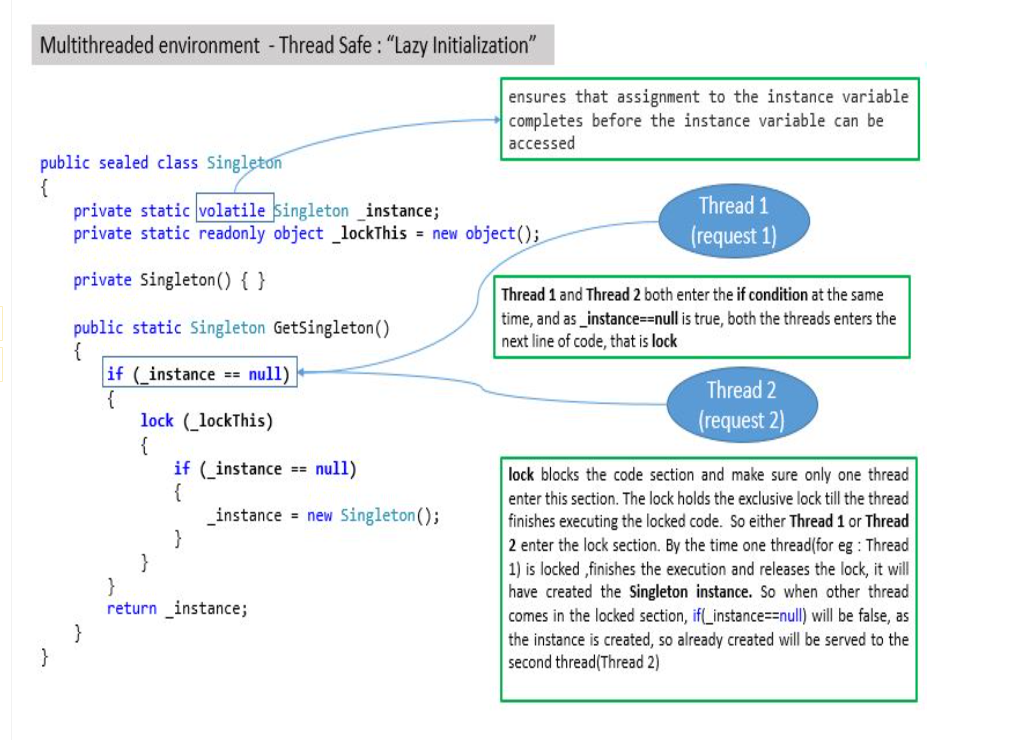
            }

        }

**return** obj;

    }

}



## 5. Bill Pugh Singleton Implementation

package com.journaldev.singleton;

public class BillPughSingleton {

private BillPughSingleton(){}

private static class SingletonHelper {

private static final BillPughSingleton INSTANCE = new BillPughSingleton();

}

public static BillPughSingleton getInstance() {

return SingletonHelper.INSTANCE;

}

}

Notice the private inner static class that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is not loaded into memory and only when someone calls the getInstance() method, this class gets loaded and creates the singleton class instance. This is the most widely used approach for the singleton class as it doesn’t require synchronization.

There are many classes in Java API designed with singleton design pattern:-

* [java.lang.Runtime#getRuntime()](http://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html#getRuntime--)
* [java.awt.Desktop#getDesktop()](http://docs.oracle.com/javase/8/docs/api/java/awt/Desktop.html#getDesktop--)

**Three ways to break singleton pattern**

1)Reflection

2)Serialization

3)Cloning.

**public** **static** **void** main(String[] args)

    {

        Singleton instance1 = Singleton.instance;

        Singleton instance2 = **null**;

**try**

        {

            Constructor[] constructors =

                    Singleton.**class**.getDeclaredConstructors();

**for** (Constructor constructor : constructors)

            {

                // Below code will destroy the singleton pattern

                constructor.setAccessible(**true**);

                instance2 = (Singleton) constructor.newInstance();

**break**;

            }

        }

**catch** (Exception e)

        {

            e.printStackTrace();

        }

    System.out.println("instance1.hashCode():- "

                                      + instance1.hashCode());

    System.out.println("instance2.hashCode():- "

                                      + instance2.hashCode());

    }

**Overcome reflection issue:** To overcome issue raised by reflection, [enums](https://www.geeksforgeeks.org/enum-in-java/) are used because java ensures internally that enum value is instantiated only once. Since java Enums are globally accessible, they can be used for singletons. Its only drawback is that it is not flexible i.e it does not allow lazy initialization.

//Java program for Enum type singleton

**public** **enum** Singleton

{

  INSTANCE;

}

**JVM handles the creation and invocation of enum constructors internally**

**2)Serialization**

**public** **class** GFG

{

**public** **static** **void** main(String[] args)

    {

**try**

        {

            Singleton instance1 = Singleton.instance;

            ObjectOutput out

                = **new** ObjectOutputStream(**new** FileOutputStream("file.text"));

            out.writeObject(instance1);

            out.close();

            // deserialize from file to object

            ObjectInput in

                = **new** ObjectInputStream(**new** FileInputStream("file.text"));

            Singleton instance2 = (Singleton) in.readObject();

            in.close();

            System.out.println("instance1 hashCode:- "

                                                 + instance1.hashCode());

            System.out.println("instance2 hashCode:- "

                                                 + instance2.hashCode());

        }

**catch** (Exception e)

        {

            e.printStackTrace();

        }

    }

}

**Overcome serialization issue:-** To overcome this issue, we have to implement method readResolve() method.

**protected** Object readResolve()

    {

**return** instance;

    }

**3)Cloning**

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

}

public class CloningSingleton {

public static void main(String[] args) throws CloneNotk,l;.SupportedException, Exception {

Singleton instanceOne = Singleton.getInstance();

Singleton instanceTwo = (Singleton) instanceOne.clone();

System.out.println("hashCode of instance 1 - " + instanceOne.hashCode());

System.out.println("hashCode of instance 2 - " + instanceTwo.hashCode());

}

}

Overcome cloning :

@Override

protected Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException();

}

Façade Design Pattern :  Its purpose is to hide internal complexity behind a single interface that appears simple on the outside.

[Linked List in Java: All You Need to Know About it (simplilearn.com)](https://www.simplilearn.com/tutorials/java-tutorial/linked-list-in-java#:~:text=Working%20Of%20Linked%20List%20in%20Java%20Is%20As,the%20next%20container%20in%20the%20list.%20More%20items)

**SinglyLinkedList implementation in Java**

Public class SInglyLinkedList{

Class Node{

Int data;

Node next;

Public Node(int data){

this.data=data

this.next=null;

}

}

Public Node head=null;

Public Node tail=null;

Public void insert(int data){

Node node=new Node(data);

//if list is empty

If(head==null){

head=node;

tail=node;

}

Else{

tail.next=node;

Tail=node;

}

}

Public void displayList(){

If(head==null){

SOP(“List is empty”);

return;

}

Node current=head;

While(current!=null){

S.O.P(current.data);

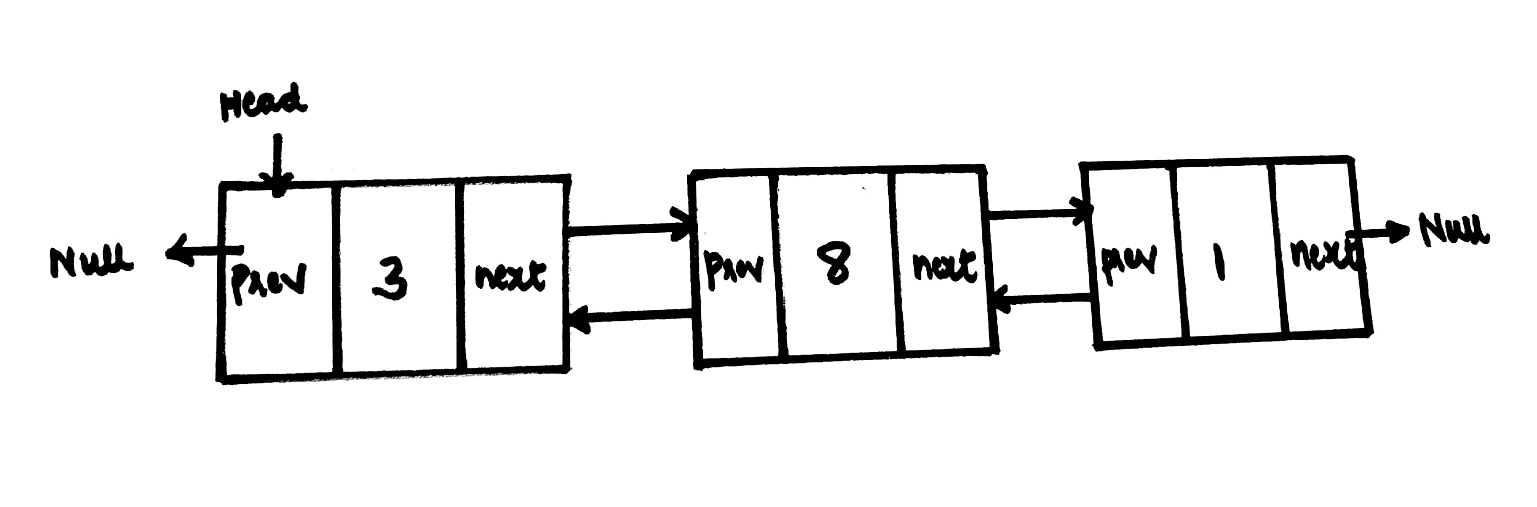
current=current.next;

}

}

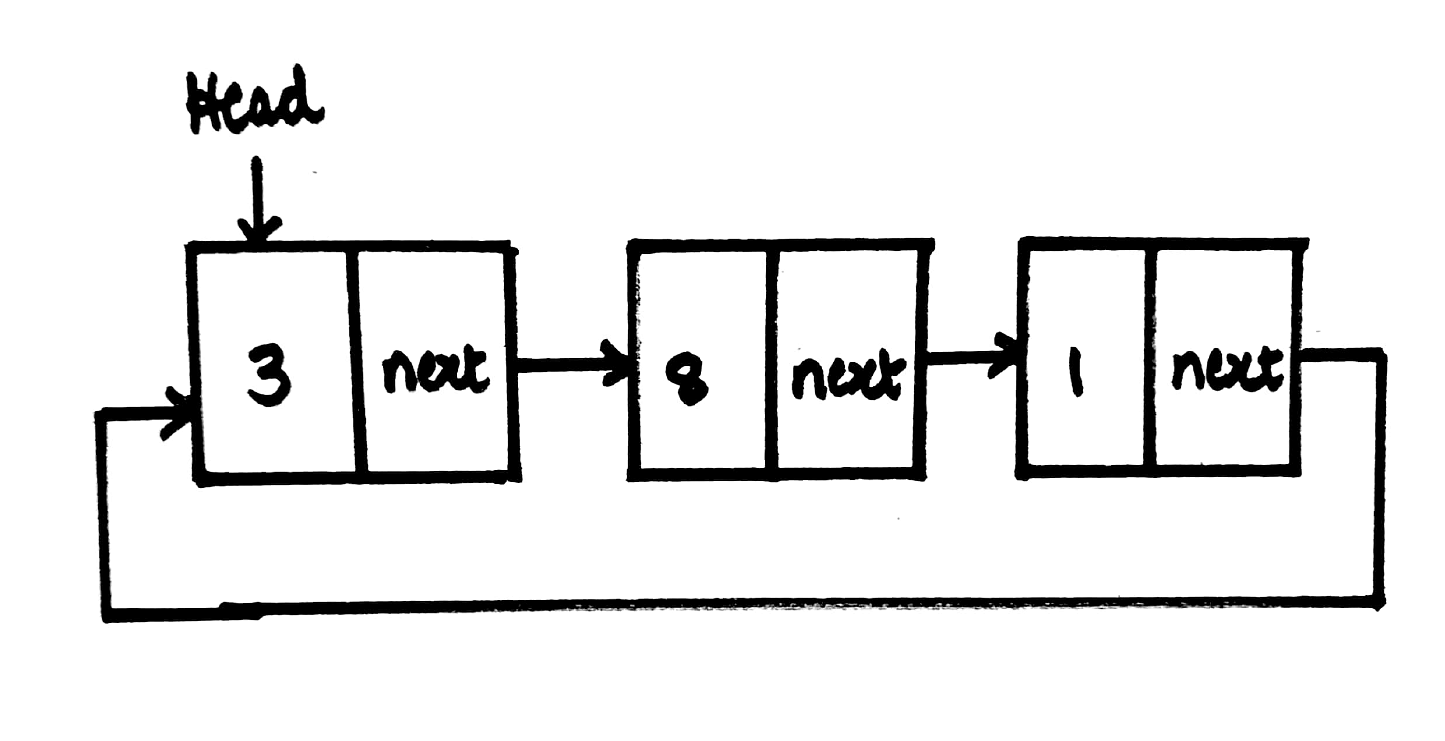
}

**Double Linked List**



* Data is the data stored in the node and each node consists of two pointers namely the previous pointer and the next pointer.
* It is used in the navigation systems where front and back navigation is required.
* It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
* Doubly Linked List is also used in constructing [**MRU**](https://www.geeksforgeeks.org/program-for-k-most-recently-used-mru-apps/)**/**[**LRU**](https://www.geeksforgeeks.org/lru-cache-implementation/)(Most/least recently used) cache.

**Circular Linked List**



It is the type of linked list consisting of a sequence of nodes where each node consists of data and a link to the next node and the last node in the list (also called as tail) that points to the first node in the list (also called as head) is called as Circular Linked List.

   tail.next = head;

* Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

**JVM :**

Similar to virtual machines, JVM creates an isolated space on host machines. This space can be used to execute programs irrespective of platform or OS of machine.

Java code is first complied into byte code to generate a class file. This class file is then interpreted by JVM for the underlying platform.

**SAGA Pattern:**

It is used to create a chain of ACID transactions across services. It is used on distributed environments that multiple services communicate.

You have two services, so you have two local transactions to accomplish a Saga.

First Service does some local ACID transaction and publishes an event to notify the second service.

Second service listens the event and does its local ACID transaction and publishes another event.

Then first service

listens the event from the second service and completes the saga.

How to make sure the local transaction plus publishing the event operation works in a consistent manner.

If you just first commit to a database transaction and then publish the event, if the publish operation

fails,

Saga cannot continue and you will leave the system in inconsistent state.

[Managing data consistency in a microservice architecture using Sagas part 2 - coordinating sagas (chrisrichardson.net)](https://chrisrichardson.net/post/sagas/2019/08/04/developing-sagas-part-2.html)

Choreography is an event-driven approach. When using choreography, there isn’t a central coordinator telling the saga participants what to do. Instead, the saga participants subscribe to each other’s events and respond accordingly. When a service updates its data, it simply emits a [domain event](https://microservices.io/patterns/data/domain-event.html) announcing what it has done. Other services subscribe to those events, which trigger updates that emit additional events. For example, in the Create Order saga, the Order Service emits an OrderCreated event, which causes the Customer Service to attempt to reserve credit. In a later blog post, I’ll describe how to implement the Create Order saga using choreography in more detail.

Orchestration uses a saga orchestrator that tells the saga participants what to do. The saga orchestrator communicates with the participants using [request/asynchronous response-style](https://microservices.io/patterns/communication-style/messaging.html) interaction. To execute a saga step, it sends a command message to a participant telling it what operation to perform. After the saga participant has performed the operation, it sends a reply message to the orchestrator. The orchestrator then processes the reply message and determines which saga step to perform next. For example, the Create Order saga orchestrator sends a Reserve Credit command to the Customer Service, which then attempts to reserve credit. In a later blog post, I’ll describe how to implement the Create Order saga using orchestration in more detail.

Each step of saga updates a database (e.g. a business object or saga orchestrator) and sends a message/event. These two actions must be done atomically in order to avoid data inconsistencies and bugs.

* [Transactional Outbox pattern](https://microservices.io/patterns/data/transactional-outbox.html) - publish a message by inserting it into an OUTBOX table as part of the database transaction. A separate process (e.g. the [Eventuate CDC](https://eventuate.io/)) retrieves the message from the OUTBOX table and sends it to the message broker.

[Transactional outbox (microservices.io)](https://microservices.io/patterns/data/transactional-outbox.html)

Instead of directly sending a message to Kafka when updating the database, the service uses a single transaction to both perform the normal update and insert the message into a specific outbox table within its database. Because this is done within a single database transaction, either the changes to the service’s model are persisted and the message gets safely stored in the outbox table, or none of these changes gets applied. Once the transaction has been written to the database’s transaction log, the Debezium change data capture(message relay) process can pick up the outbox message from there and send it to Apache Kafka.

Chart

Description automatically generated

 the same outbox message could be sent to Kafka multiple times. To allow consumers to detect and ignore duplicate messages, each message should have a unique id. This could for instance be a UUID or a monotonically increasing sequence specific to each message producer, propagated as a Kafka message header.

**Multiton Design Pattern:**

public enum NazgulName {

KHAMUL, MURAZOR, DWAR, JI\_INDUR, AKHORAHIL, HOARMURATH, ADUNAPHEL, REN, UVATHA

}

public final class Nazgul {

private static final Map<NazgulName, Nazgul> nazguls;

private final NazgulName name;

static {

nazguls = new ConcurrentHashMap<>();

nazguls.put(NazgulName.KHAMUL, new Nazgul(NazgulName.KHAMUL));

nazguls.put(NazgulName.MURAZOR, new Nazgul(NazgulName.MURAZOR));

nazguls.put(NazgulName.DWAR, new Nazgul(NazgulName.DWAR));

nazguls.put(NazgulName.JI\_INDUR, new Nazgul(NazgulName.JI\_INDUR));

nazguls.put(NazgulName.AKHORAHIL, new Nazgul(NazgulName.AKHORAHIL));

nazguls.put(NazgulName.HOARMURATH, new Nazgul(NazgulName.HOARMURATH));

nazguls.put(NazgulName.ADUNAPHEL, new Nazgul(NazgulName.ADUNAPHEL));

nazguls.put(NazgulName.REN, new Nazgul(NazgulName.REN));

nazguls.put(NazgulName.UVATHA, new Nazgul(NazgulName.UVATHA));

}

private Nazgul(NazgulName name) {

this.name = name;

}

public static Nazgul getInstance(NazgulName name) {

return nazguls.get(name);

}

public NazgulName getName() {

return name;

}

}

[Securing Spring Boot microservices with OAuth2 | Talentify](https://talentify.com/securing-spring-boot-microservices-with-oauth2/) good

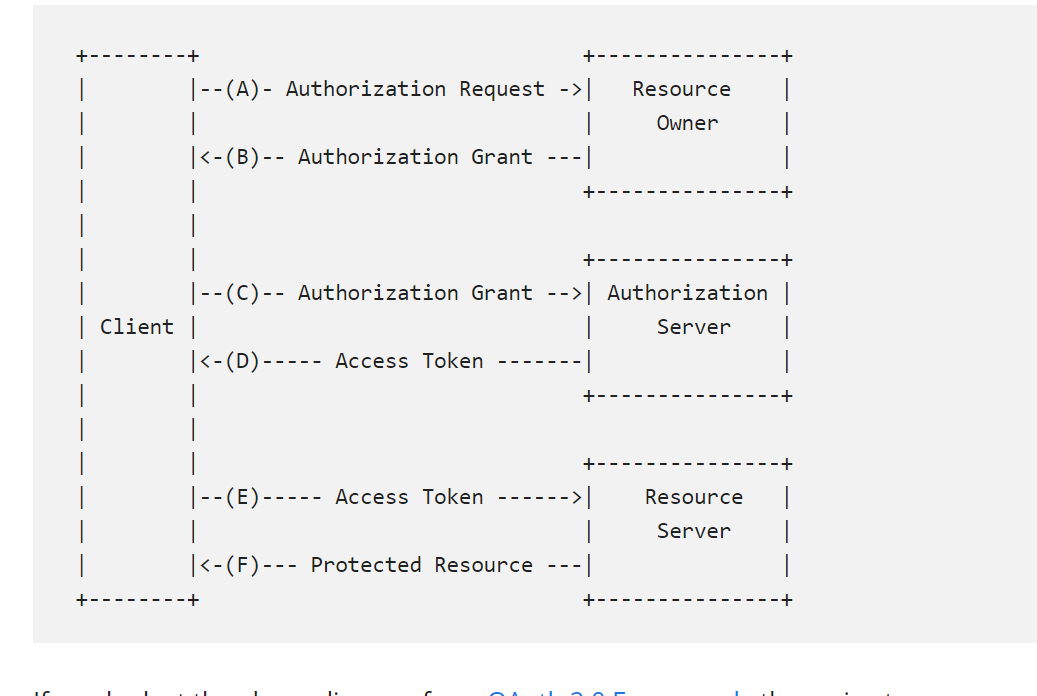
[Building Microservices Using Spring Boot and Securing Them With OAuth and OpenID - Part 1 - DZone Integration](https://dzone.com/articles/building-microservice-using-spring-boot-and-secure)- good

[Building Microservices Using Spring Boot and Securing Them With OAuth and OpenID - Part 2 - DZone Integration](https://dzone.com/articles/building-microservices-using-spring-boot-and-secur)

[Spring Boot Security Auto-Configuration | Java Development Journal (javadevjournal.com)](https://www.javadevjournal.com/spring-boot/spring-boot-security-auto-configuration/#:~:text=Spring%20Boot%20come%20with%20a%20rich%20set%20of,to%20determine%20whether%20to%20use%20httpBasic%20or%20formLogin.)

[Securing Spring Boot Microservices with JSON Web Tokens (JWT) - DZone Microservices](https://dzone.com/articles/securing-spring-boot-microservices-with-json-web-t)

 To protect the endpoint each microservice should get an identity and a set of permissions so that the API endpoint security component is able to verify both.



Exception Translation in spring DAO layer @Repository

One important change that needs to be written while using Spring DAO is, that it has to be annotated with **@Repository**. The reason for doing this is the exceptions that may arise in the underlying technology like JDBC, Hibernate, JPA, etc. are consistently translated into their respective DataAccessException subclass

Instead, if it is annotated with @Repository, then the exceptions related to the current underlying technologies will be directly translated to the spring DataAccessException. Because of this feature, though the underlying technologies are changed from hibernate to JPA or from JPA to hibernate, then the same Spring DataAccessExceptions will still be thrown. According to the underlying technologies, the spring will translate according to their native exceptions

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

The main difference between the @restcontroller and the @controller is that the @restcontroller combination of the @controller and @ResponseBody annotation.

[What is the difference between Clustered and Non-Clustered Indexes in SQL Server? (sqlshack.com)](https://www.sqlshack.com/what-is-the-difference-between-clustered-and-non-clustered-indexes-in-sql-server/)

**Secure REST APIs:**

### Transport Layer Security

### Rate Limiting

Rate limiting is typically implemented in an API gateway or dedicated protection product (such as a WAF or API firewall).

**OATH2**

**API KEYS**

Kafka Consumer API is not thread safe. ConcurrentKafkaListenerContainerFactory api provides concurrent way of using Kafka Consumer API along with setting other kafka consumer properties.

Print manager’s name for each employee

SELECT emp.EmpId EmployeeId, emp.EmployeeName EmployeeName,

emp.Manager\_Id ManagerId, ISNULL(mng.EmployeeName, 'No Boss') AS ManagerName

FROM TblEmployeeManager emp

LEFT JOIN TblEmployeeManager mng

ON emp.Manager\_Id = mng.EmpId

[Java semaphore producer consumer example (demo2s.com)](https://www.demo2s.com/java/java-semaphore-producer-consumer-example.html)