

IN-MEMORY MULTI-THREADED DATABASE MVC WEB APP

Abstract

In this project, I've built an in-memory database mvc web app, where the server-side is connected to servlets & jsp and the client-side is a user-interface on the web, with keeping in mind synchronizing accessing shared resource since the app is meant to be multi-threaded. Also, the system is connected to a simple CI/CD Jenkins server. In this report, I'll be defending my project against clean code points, SOLID principles, Effective java points, ACID criteria, design patterns and synchronization.

***Disclaimer:** All of the code snapshots used in this document are taken from my project.

Table of Contents

.....	0
Abstract.....	1
Clean Code	5
I. Meaningful Names.....	5
II. Simple Functions.....	6
III. Don't Repeat Yourself	7
IV. Comments.....	7
V. Data Abstraction.....	8
VI. Don't Return Null.....	9
Effective Java	10
1. Item 4: Enforce non-instantiability with a private constructor	10
2. Item 5: Prefer dependency injection to hardwiring resources	10
3. Item 6: Avoid creating unnecessary objects	11
4. Item 7: Eliminate obsolete object references	11
5. Item 8: Avoid finalizers and cleaners	11
6. Item 9: Prefer try-with-resources to try-finally	12
7. Item 12: Always override toString	12
8. Item 15: Minimize the accessibility of classes and members.....	13
9. Item 16: In public classes, use accessor methods, not public fields	13
10. Item 17: Minimize mutability	14
11. Item 20: Prefer interfaces to abstract classes	14
12. Item 21: Design interfaces for posterity	15
13. Item 22: Use interfaces only to define types	15
14. Item 40: Consistently use the Override annotation	15
15. Item 49: Check parameters for validity.....	16
16. Item 51: Design method signatures carefully.....	16
17. Item 54: Return empty collections or arrays, not nulls	17
18. Item 57: Minimize the scope of local variables	17
19. Item 58: Prefer for-each loops to traditional for loops:.....	17
20. Item 61: Prefer primitive types to boxed primitives.....	18

21. Item 63: Beware the performance of string concatenation.....	18
22. Item 64: Refer to objects by their interfaces	18
23. Item 68: Adhere to generally accepted naming conventions	19
24. Item 69: Use exceptions only for exceptional conditions	19
25. Item 72: Favor the use of standard exceptions	19
26. Item 77: Don't ignore exceptions	20
27. Item 78: Synchronize access to shared mutable data	20
28. Item 79: Avoid excessive synchronization	21
29. Item 83: Use lazy initialization judiciously	21
SOLID Principles	22
I. (SRP): Single Responsibility Principle	22
II. (OCP): Open Close Principle	22
III. (LSP): Liskov Substitution Principle	23
IV. (ISP): Interface Segregation Principle	23
V. (DIP): Dependency Inversion Principle	24
Design Patterns.....	25
I. Singleton	25
II. Factory Method	26
III. Null Object.....	27
ACID Criteria.....	28
I. Atomicity.....	28
II. Consistency.....	28
III. Isolation.....	30
IV. Durability.....	31
System's Architecture	32
I. (MVC): Model View Controller	32
II. UML Class Diagrams.....	33
System's Data structures	37
I. Cache & Cache Item.....	37
II. Reentrant Lock.....	39
Synchronization	40
CI/CD.....	42
I. CI: Continuous Integration	42

1. Build	42
2. Test.....	42
II. CD: Continuous Delivery/Deployment.....	43
1. Acceptance Test	43
2. Deploy to Staging	43
3. Deploy to Production	43
Jenkins.....	44

Clean Code

I. Meaningful Names

Names should be clear, descriptive and strait to the point. Your naming pattern should rely on a main important idea, which is revealing the intention. Each module in your system whether it is tiny or huge, should be fully understood by reading its name only. The name of a variable, function or class should show what it does, why it exists and how it is used. Moreover, picking up good names help you to avoid writing worthless comments, where just a clear name can replace that smelly comment.

Always try to look for the naming conventions of a particular module, there is a high chance that experts and their communities have agreed on a clear, accurate naming convention of a particular module before you. Why inventing a new naming convention when there are a lot well known and agreed on names? Using naming conventions can save you from explaining your naming strategy to your colleagues. To sum it up, just stick to the naming conventions.

Here are some examples of good naming from my project:

```
public static void chooseTable(int tableIndex){
```

As shown in the snapshot above, the method name can answer you about what it does and how. Also, it is worth mentioning that methods should be named in a camel case style by convention.

```
public class DatabaseCache extends Cache {
```

As shown in the snapshot above, the class name reveals to the reader its intent of use. Classes names use pascal case style by convention.

```
Entity recordToDelete
```

As shown in the snapshot above, the variable name is simple, clear and strait to the point. By the way, variables names use camel case by convention.

II. Simple Functions

Functions should be short, clear and responsible for a single mission. It is always preferred that your functions don't exceed 30 lines of code. Long functions tend to make you forget about its functionality at the first place. Moreover, the shorter the function it is, the easier it is to be understood. Usually, long functions are not single responsible, which add a lot of complexity to it and make it a bugs-magnet, and yet hard to understand. On the long run, well built functions show you its beauty when there is a bug and it's tracked by seconds, whereas complex functions will make you lost in its code and you'll end up spending hours troubleshooting it when maintenance is required.

As discussed in the previous section, functions should have descriptive name to answer what it does and why it is used. Also, the number of parameters should be short, the ideal number is zero and not more than 4 parameters.

Here are some examples of simple functions from my project:

```
public static int getTableIndex(String table){  
    if(table.equalsIgnoreCase(ENTITY1))  
        return 0;  
    else  
        return 1;  
}
```

```
@Override  
public void freeCache() { cachedItems.remove(getLFU()); }
```

```
public void deleteByKey(int key) { cachedItems.remove(key); }
```

As shown above, all of the functions are short to read, simple to understand and easy to debug. It is worth mentioning that they all are single responsible. Each of them is doing a single, specific task.

III. Don't Repeat Yourself

When you write the same block of code more than once, then you are repeating yourself, which is quite wrong. You should put a repeated block of code in a function and call it when needed instead of writing duplicates of a code block. In a good programming world, doing this is prohibited. Functions are created to be called and code blocks should be written once, not more.

What if you discovered a bug in a block of code, which you've made duplicates of? It doesn't make any sense to go to each duplicate and do the same exact modification to fix the bug. To sum up, redundant code should be eliminated and refactored to a single method so it can be called whenever needed.

IV. Comments

A Comment is a misunderstood part in programming. Some programmers tend to write a comment for each small tiny block of code. They think that it is good and free, then why not use it? As a good programmer, you should have an eye that can determine whether writing a comment for a block of code is needed or not. Often, comments are just cluttering the code. It is better to simplify your code and make it easy to read. Writing a comment to a clean, simple, single responsible block of code isn't needed in the most cases. Just make your code explain itself by following good programming conventions and habits.

If you face bad code, which cannot be understood without a comment, then it is better to refactor or rewrite it than adding a new smelly comment. Try to make the Scout Boy strategy as a habit for you when you review chunks of code. That strategy is all about not leaving a place without doing enhancements in it. Just measure the enhancement, and if it's worth doing as well as you have a time for doing so then just do it.

Here is an example from my project of a simple method that expresses itself:


```

@Override
public void setAttributes(String[] attributes) {
    lock.lock();
    setId(Integer.parseInt(attributes[0]));
    setName(attributes[1]);
    setCountry(attributes[2]);
    setPhone(Integer.parseInt(attributes[3]));
    setEmail(attributes[4]);
    lock.unlock();
}

```

As shown in the snapshot, the method itself is easy to understand and simple. Some programmer might add a comment for the same block of code that says “This method is to set the attributes of a record” but adding clutter to as clear piece of code as that is wrong.

V. Data Abstraction

As good programmers, we always tend to hide our variables by making its access modifier of type private. The reason behind that is we don't want anyone to rely on them. Relying on concrete objects is wrong. It limits the extendibility of your code. Your code should be closed for modification but open for extending. Therefore, the best way to achieve an extendible code is to use abstraction layers. Abstraction is all about looking to the full picture in general, instead of looking in the details. This concept alone can make your code far more extendible.

Here is an example from my project of an abstract layer:

```

public abstract class CacheItem {
    private int hitCount = 0;
    public int getHitCount() { return hitCount; }

    public void incrementHitCount() { this.hitCount++; }
}

```

As shown above in the snapshot, this abstract class is acting as an abstraction layer, so consumers can extend their projects by implementing it whenever they need.

By the way, interfaces and abstract classes can act as an abstract layer in your system. What they do is forcing any other module, which implements them, to a contract of implementing their methods. Interfaces are preferable over abstract classes, as they are purely act as an abstract layer.

VI. Don't Return Null

When you make your method return null at a specific condition, then you're basically making a bug or causing other module to crash at any time. If the other module has missed just a check statement of nullity, then a big bomb of crashes will bang in front of his face. To sum it up, returning null is dangerous and you should find an alternative to that bad habit.

One of the solutions is to return a null object instead of returning null. For string, you can return an empty string, and for list or arrays, you can return them empty. Although you can say that creating a null object can affects performance, but you can solve this leak by making a static null object and keep returning the exact object when it's needed to do so. By the way, this solution is called "Null Object Pattern".

Here is an example from my project of a null object pattern:

```
public class NullEntity extends Entity {
    public static final NullEntity nullEntity = new NullEntity();
    private NullEntity(){}

    public static NullEntity getInstance() { return nullEntity; }
    @Override
    public void setAttributes(String[] attributes) {
        //This method is unsupported
        throw new UnsupportedOperationException();
    }
}
```

As shown above in the code snippet, there is a class called NullEntity, and it is used basically to return an instance of a null object when the situation requires to return null. If the consumer missed to check the nullity of the object and tried to use some functionality on it, it will throw to him an exception that this object doesn't support that functionality. He will then know that this isn't the expected object and will check the returned object.

Effective Java

1. Item 4: Enforce non-instantiability with a private constructor

- Static classes, which groups only static members or fields are used as a utility to provide some help, they aren't meant to be instantiated. unintentionally they can be instantiated. To prevent that you should write a private parameter-less constructor. Because a default constructor is always there and the only way to take it away is by writing another constructor. Also, this solution is better than making a static class abstract, since it could be sub-typed and hence instantiating the sub-class. Moreover, it misleads the consumer of the api that this static class should be inherited, which is totally wrong.

```
public class OperationsUtil {  
    private static final EntityFactory entityFactory = EntityFactory.getEntityFactory();  
  
    private OperationsUtil(){}  
  
    public static void chooseTable(int tableIndex){  
        if(tableIndex == 0){  
            table = TABLE1;  
            entity = ENTITY1;  
        }  
        else{  
            table = TABLE2;  
            entity = ENTITY2;  
        }  
    }  
}
```

2. Item 5: Prefer dependency injection to hardwiring resources

- Do not use a singleton or static utility class to implement a class that depends on one or more underlying resources whose behavior affects that of the class, and do not have the class create these resources directly. Instead, pass the resources, or factories to create them, into the constructor (or static factory or builder). This practice, known as dependency injection, will greatly enhance the flexibility, reusability, and testability of a class.

3. Item 6: Avoid creating unnecessary objects

- Use static factory methods.
- Use utility classes.
- Reuse immutable objects.

```
public class DAOFactory {  
    private DAOFactory(){}  
  
    public static DAO getInstanceByType(String type){  
        if(type.equalsIgnoreCase( anotherString: "customer"))  
            return new CustomerDAO();  
        else  
            if(type.equalsIgnoreCase( anotherString: "account"))  
                return new AccountDAO();  
            return new NullDAO();  
    }  
}
```

4. Item 7: Eliminate obsolete object references

- When shrink down a data structure of items (array, stack, etc.), always make items in the data structure reference to null when you've done with them (for a stack, pop is not sufficient), gc cannot decide whether they are being used (active portion) or not (inactive portion). For gc, all of the elements are in the data structure, only the programmer know that they aren't any more.
- Generally speaking, whenever a class manages its own memory, the programmer should be alert for memory leaks. Whenever an element is freed, any object references contained in the element should be nulled out.

5. Item 8: Avoid finalizers and cleaners

- Implement AutoCloseable interface to increase the performance up to 50x

- If object has not been constructed correctly, using finalizers or cleaners will cause random un-expected bugs as it works on corrupted objects.

6. Item 9: Prefer try-with-resources to try-finally

- try-finally is ugly when used with more than one resource!
- You should consider implementing `AutoCloseable` when your class represents a resource to take benefit of try-with-resource
- Try-finally shows only the stack trace of the last exception, whereas try-with-resource shows the first one and suppress the next ones but it shows them and mark them as suppressed exceptions
- Exceptions are more useful and it make the code short, clean and clear.

7. Item 12: Always override toString

- The general contract for `toString` says that the returned string should be “a concise but informative representation that is easy for a person to read.”
- Default `toString` in objects return the class name, followed by an '@' sign, which is followed by the hexa-decimal representation of the hash code. It can be concise and easy to read for a skilled java developer, but it is not very informative.

```
@Override
public String toString() {
    return getId() + "," + getCustomerId() + "," + getUsername() + "," + getPassword();
}
```

- It makes no sense to write a `toString` method in a static utility class
- Implement it in every instantiable class you write, unless a superclass has already done so. It makes classes much more pleasant to use and aids in debugging. The `toString`

method should return a concise, useful description of the object, in an aesthetically pleasing format.

8. Item 15: Minimize the accessibility of classes and members

- A well-designed component hides all its implementation details, cleanly separating its API from its implementation.
- This concept, known as information hiding or encapsulation, is a fundamental tenet of software design.
- It decouples the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation.
- Make each class or member as inaccessible as possible
- Instance fields of public classes should rarely be public
- Classes with public mutable fields are not generally thread-safe (Race condition problem).

9. Item 16: In public classes, use accessor methods, not public fields

- If data fields of a class are accessed directly, this class do not offer the benefits of encapsulation

```
public class Account extends Entity {  
    private int id;  
    private int customerId;  
    private String username;  
    private String password;  
  
    public int getId() { return id; }  
    public int getCustomerId() { return customerId; }  
    public String getUsername() { return username; }  
    public String getPassword() { return password; }
```

- If a class is accessible outside its package, provide accessor methods
- While it's never a good idea for a public class to expose fields directly, it is less harmful, though still questionable, if the fields are immutable and sacrificing flexibility is worth it.

10. Item 17: Minimize mutability

- An immutable class is simply a class whose instances cannot be modified
- Immutable classes are easier to design, implement, and use than mutable classes. They are less prone to error and are more secure.
- Don't provide methods that modify the object's state (known as mutators).
- Ensure that the class can't be extended by making it final.
- Make all fields final and private
- Immutable objects are inherently thread-safe; they require no synchronization
- Immutable objects provide failure atomicity for free

11. Item 20: Prefer interfaces to abstract classes

- A class must be a subclass of the abstract class. Because Java permits only single inheritance.
- Any class that defines all the required methods and obeys the general contract is permitted to implement an interface, regardless of where the class resides in the class hierarchy.
- Existing classes can easily be retrofitted to implement a new interface

12. Item 21: Design interfaces for posterity

- Adding new methods to existing interfaces is fraught with risk.
- There is no guarantee that added methods will work in all preexisting implementations.
- It is still of the utmost importance to design interfaces with great care.
- While it may be possible to correct some interface flaws after an interface is released, you cannot count on it.

13. Item 22: Use interfaces only to define types

- A class implements an interface should say something about what a client can do with instances of the class.
- They should not be used merely to export constants.
- Export the constants with a non-instantiable utility class.

```
public interface DAO {  
    void setAttributeValue(Entity row, String attribute, String newValue);  
}
```

14. Item 40: Consistently use the Override annotation

- It indicates that the annotated method declaration overrides a declaration in a supertype.
- It will protect you from a large class of nefarious bugs.
- IDE will generate a warning if you have a method that doesn't have an Override annotation but does override a superclass method.
- It is good practice to use Override on concrete implementations of interface methods to ensure that the signature is correct.

15. Item 49: Check parameters for validity

- You should attempt to detect errors as soon as possible after they occur. Failing to do so makes it less likely that an error will be detected and makes it harder to determine the source of an error once it has been detected.
- If an invalid parameter value is passed to a method and the method checks its parameters before execution, it will fail quickly and cleanly with an appropriate exception.
- The method could fail with a confusing exception in the midst of processing. Worse, the method could return normally but silently compute the wrong result.

```
public void setId(int id) throws IllegalArgumentException {  
    if(id < 0)  
        throw new IllegalArgumentException();  
    else  
        this.id = id;  
}
```

16. Item 51: Design method signatures carefully

- Choose method names carefully.
- Don't go overboard in providing convenience methods.
- Avoid long parameter lists. Aim for four parameters or fewer.
- For parameter types, favor interfaces over classes

17. Item 54: Return empty collections or arrays, not nulls

- Returning null in place of an empty collection or array is error-prone and has no performance advantages.
- Programmer writing the client might forget to write the special-case code to handle a null return.

```
return NullEntity.getInstance();
```

18. Item 57: Minimize the scope of local variables

- Increase the readability and maintainability of your code and reduce the likelihood of error.
- Try to declare a local variable where it is first used.
- If a variable is declared before it is used, it's just clutter (noise).
- If a variable is declared outside of the block in which it is used, it remains visible after the program exits that block.
- Nearly every local variable declaration should contain an initializer.
- Prefer for loops to while loops.
- Keep methods small and focused.

19. Item 58: Prefer for-each loops to traditional for loops:

- The iterator and the index variables are both just clutter
- It offers clarity, flexibility, and bug prevention, with no performance penalty.
- It can iterate over any iterable collection.

20. Item 61: Prefer primitive types to boxed primitives

- Autoboxing and Auto-unboxing blur but do not erase the distinction between the primitive and boxed primitive types, they are different.
- Primitives are more time- and space-efficient than boxed primitives.
- Boxing primitive values can result in costly and unnecessary object creations.
- Comparing two boxed primitives with the == operator does an identity comparison, which is almost certainly not what you want.

```
private static final String TABLE1 = "customersTable.csv";
```

21. Item 63: Beware the performance of string concatenation

- Using the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n.
- Strings are immutable. When two strings are concatenated, the contents of both are copied.
- To achieve acceptable performance, use a StringBuilder in place of a String, then start appending them using StringBuilder's append method.

```
output.append(recordFromDatabase).append(System.LineSeparator());
```

22. Item 64: Refer to objects by their interfaces

- If appropriate interface types exist, then parameters, return values, variables, and fields should all be declared using interface types.
- Using interfaces as types, program will be much more flexible.
- If there is no appropriate interface, just use the least specific class in the class hierarchy that provides the required functionality.

23. Item 68: Adhere to generally accepted naming conventions

- Packages, classes, interfaces, methods, fields, and type variables all have handful typographical naming conventions. You should rarely violate them and never without a very good reason.
- Violations have the potential to confuse and irritate other programmers who work with the code and can cause faulty assumptions that lead to errors.

24. Item 69: Use exceptions only for exceptional conditions

- Because exceptions are designed for exceptional circumstances, there is little incentive for JVM implementors to make them as fast as explicit tests.
- Placing code inside a try-catch block inhibits certain optimizations that JVM implementations might otherwise perform.
- The exception-based idiom is far slower than the standard one.
- A well-designed API must not force its clients to use exceptions for ordinary control flow.

25. Item 72: Favor the use of standard exceptions

- Reusing standard exceptions has several benefits.
- It makes your API easier to learn and use because it matches the established conventions that programmers are already familiar with.
- Programs using your API are easier to read because they aren't cluttered with unfamiliar exceptions.
- Fewer exception classes mean a smaller memory footprint and less time spent loading classes.

- Do not reuse Exception, RuntimeException, Throwable, or Error directly. Treat them as if they were abstract.

26. Item 77: Don't ignore exceptions

- An empty catch block defeats the purpose of exceptions.
- If you choose to ignore an exception, the catch block should contain a comment explaining why it is appropriate to do so, and the variable should be named ignored.

27. Item 78: Synchronize access to shared mutable data

- Proper use of synchronization guarantees that no method will ever observe the object in an inconsistent state.
- It ensures that each thread entering a synchronized method or block sees the effects of all previous modifications that were guarded by the same lock.
- Synchronization is required for reliable communication between threads as well as for mutual exclusion.

```
@Override
public void setAttributes(String[] attributes) {
    lock.lock();
    setId(Integer.parseInt(attributes[0]));
    setCustomerId(Integer.parseInt(attributes[1]));
    setUsername(attributes[2]);
    setPassword(attributes[3]);
    lock.unlock();
}
```

28. Item 79: Avoid excessive synchronization

- Excessive synchronization can cause reduced performance, deadlock.
- To avoid liveness and safety failures, never cede control to the client within a synchronized method or block.
- You should do as little work as possible inside synchronized regions.
- Synchronizing blocks are always preferred over synchronizing methods.

29. Item 83: Use lazy initialization judiciously

- Under most circumstances, normal initialization is preferable to lazy initialization.
- It violates keeping variables in the tightest scope of use.

SOLID Principles

One of the most if not the most important concept in software development is the SOLID principles. The word SOLID is constructed by taking the first letter in each of the six design principles, which are:

I. (SRP): Single Responsibility Principle

A class should have only one reason to change (responsible for a one actor). Each method in your class & each class in your module should be doing one & only one thing!

Simplicity would be at your fingertips whenever you stick to this principle, just look at this snippet of code from my system, can't you imagine how simple it is? I won't say a word about its responsibility.

Actually, you are capable of determining the functionality of it as you were the original writer of the code snippet!

```
@Override  
public void freeCache() { //Eviction Policy: LFU  
    cachedItems.remove(getLFU());  
}
```

II. (OCP): Open Close Principle

Classes should be open for extension, but closed for modifications.

Your modules should be extendable, not modifiable. Let me explain it to you with a code snippet.

```
public class Operations implements Read, Insert, Update, Delete
```

Just take look on the code snippet. Simply, it is saying that our operations class will have its ability to perform the CRUD operations on its own, by its own style, version, whatever ... you've got it. Imagine that there is a new pack of operations in the market and your boss asked you to support them in your system. If your module is built to be modifiable, then you'll end up missing with

hundreds, if not thousands of lines of code, and guess what, you will break your system without knowing that! On the other hand, if your module were designed in an extendable way, then you can add any new feature to the system with ease, just implement your own final version of it. Do not modify it! Just write the best, the simplest version of it.

III. (LSP): Liskov Substitution Principle

if z is subtype of Z, then all objects of type Z could be replaced with objects of type z (subtypes should be able to do what base types do).

It makes more sense that children are capable of doing what their parents do, so if your father is a programmer and can write simple code, it is more likely that you'll end up having the skill which he had! Enough chattering and let's bring a code snippet.

```
public class DatabaseCache extends Cache {  
  
    @Override  
    public void freeCache() { //Eviction Policy: LFU  
        cachedItems.remove(getLFU());  
    }  
}
```

Since the database cache is a subtype of a cache, it should support what any cache support. It should provide us with the feature of freeing it for some space when it gets out of space since it's called a cache!

IV. (ISP): Interface Segregation Principle

Subtypes shouldn't be forced to inherit useless methods.

what is the point of buying yourself useless heavy tools, which you'll be abandoning right after buying them. Your classes shouldn't be forced to inherit useless methods too! So, try to separate the parent class & the child class with an abstraction layer if the child class might not use some of the parent's methods.

V. (DIP): Dependency Inversion Principle

Big modules shouldn't depend on small modules; they both should depend on abstract instead of concrete.

It is non-sense to design your house based on the shape of the windows, which you've bought from IKEA on sales! You should pick up some windows which can fit rightly in your house, not the other around. How much windows can cost in compare with a full-fledged house, make sense now? By the way, you can achieve this principle using polymorphism.

```
@Override  
public void setAttributeValue(Entity row, String attribute, String newValue) {
```

As you can see in the code snippet, this method is depending on an entity, it doesn't care about the type of that entity. Instead, all what it cares about is just passing entity to it!

Design Patterns

I. Singleton

Singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system. Also, it can be used with the Null object or Factory method patterns to ensure that the same object is retrieved always.

Singleton pattern can be achieved by two ways, the first way is by pre-initializing a static object of the class and making the constructor private and making a method to get the same initialized instance, whereas the second way is done using lazy initialization, and the rest of it is nearly identical to the first way.

I've used this design pattern with pre-initialization in the NullEntity, CacheFactory and EntityFactory classes.

```
public class NullEntity extends Entity {  
    public static final NullEntity nullEntity = new NullEntity();  
    private NullEntity(){}  
  
    public static NullEntity getInstance() { return nullEntity; }
```

```
public class CacheFactory {  
    private static CacheFactory cacheFactory;  
    private CacheFactory(){}  
  
    public static CacheFactory getCacheFactory(){  
        if(cacheFactory == null)  
            cacheFactory=new CacheFactory();  
        return cacheFactory;  
    }  
}
```

```
public class EntityFactory {  
    private static EntityFactory entityFactory;  
  
    private EntityFactory() {  
    }  
  
    public static EntityFactory getEntityFactory() {  
        if (entityFactory == null)  
            entityFactory = new EntityFactory();  
        return entityFactory;  
    }  
}
```

II. Factory Method

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object. It eliminates the use of constructors. It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code.

Factory method pattern can be achieved by adding a method that takes the type of the needed object and returning it. It is as simple as that. You can also combine this pattern with the Singleton pattern to restrict the instantiation of the class that contains the factory method to a single object.

I've used this design pattern to manage the creation of the entities & caches whenever they're needed. By using this design pattern, you can specify the type of the entity or the type of the cache needed with ease!

```
public class CacheFactory {  
    private static CacheFactory cacheFactory;  
    private CacheFactory(){}  
  
    public Cache getCacheByType(String type, int size){  
        if(type.equalsIgnoreCase( anotherString: "database"))  
            return new DatabaseCache(size);  
        return new NullCache();  
    }  
}
```

```
public class EntityFactory {  
    private static EntityFactory entityFactory;  
    private EntityFactory() {}  
  
    public Entity getEntityByType(String type) {  
        if(type.equalsIgnoreCase( anotherString: "customer"))  
            return new Customer();  
        else if(type.equalsIgnoreCase( anotherString: "account"))  
            return new Account();  
        return NullEntity.getInstance();  
    }  
}
```

III. Null Object

A null object is an object with no referenced value or with defined neutral behavior. The null object design pattern describes the uses of such objects and their behavior. It can help preventing null pointer exception or awkward actions. Also, it supports the clean code point of not returning null. So, when you need to return null, just return a null object, which can warn the consumer that the returned object cannot be consumed by throwing a proper exception.

Null object pattern can be achieved by making a sub-type from the same parent of the often-consumed object and handling its method by describing that they aren't supported in this null object.

I've used this design pattern to prevent odd actions from happening when entities or caches are used. Just imagine that you've provide the cache factory with a missing type be created. Instead of throwing exceptions or doing some odd actions, it will just tell you that this object is null, it doesn't even exist!

```
public class NullEntity extends Entity {
    public static final NullEntity nullEntity = new NullEntity();
    private NullEntity(){}

    public static NullEntity getInstance() { return nullEntity; }
    @Override
    public void setAttributes(String[] attributes) {
        //This method is unsupported
        throw new UnsupportedOperationException();
    }

    @Override
    public void setDatabaseEntity(Entity databaseEntity) {
        //This method is unsupported
        throw new UnsupportedOperationException();
    }
}
```

ACID Criteria

ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other mishaps. In the context of databases, a sequence of database operations that satisfies the ACID properties (which can be perceived as a single logical operation on the data) is called a transaction. For example, a transfer of funds from one bank account to another, even involving multiple changes such as debiting one account and crediting another, is a single transaction.

I. Atomicity

Transactions are often composed of multiple statements. Atomicity guarantees that each transaction is treated as a single "unit", which either succeeds completely, or fails completely: if any of the statements constituting a transaction fails to complete, the entire transaction fails and the database is left unchanged.

As for my system, My CRUD operations are done on single records only, so it is not required for my case. Every operation in my system is atomic and cannot be partially failed.

II. Consistency

This property ensures that a transaction can only bring the database from one valid state to another, maintaining database invariants: any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof. This prevents database corruption by an illegal transaction, but does not guarantee that a transaction is correct. Referential integrity guarantees the primary key – foreign key relationship.

As for achieving this property, both of the databases in my system contains a primary key constraint, which is the id of the entity. Inserting a record with an existing id is a violation of the uniqueness rule of the primary key.

Also, my system will prevent you from keeping the primary key value null, as this violates the not null constraint of the primary key.

```
if (newColumnValue[0].equals("id")) // ID should remain unique  
    return false;
```

The snapshot above is taken from the update operation method body. It shows how the system cannot change the id as it should remain unique as well as not null.

```
Entity similarRecord = OperationsUtil.searchRecords(tableIndex, delimiter: "id=" + recordId, databaseCache);  
if (similarRecord.equals(NullEntity.getInstance())) {  
    return false;  
}
```

Moreover, this snapshot is taken from the insert operation method body. User cannot insert a record with an existed id. This helps preventing the inserted record from violating the uniqueness constraint of the primary key.

III. Isolation

Transactions are often executed concurrently (e.g., multiple transactions reading and writing to a table at the same time). Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially. Isolation is the main goal of concurrency control. Depending on the method used, the effects of an incomplete transaction might not even be visible to other transactions.

As for achieving this property, my system always makes an object in the cache before doing the insert operation. Whereas it retrieves an object of the record from the cache if the case was cache hit or miss for the update or delete operations. This object will be the record that will be extracted to the cache and the database. After creating or retrieving the object, then it uses the set attributes method of the object which uses the setters to set the attributes of the record for doing the operation. The critical section for the system is the body of the set attributes method. I've used reentrant locks inside that method to prevent concurrent threads from accessing it.

```
@Override
public void setAttributes(String[] attributes) {
    lock.lock();
    setId(Integer.parseInt(attributes[0]));
    setCustomerId(Integer.parseInt(attributes[1]));
    setUsername(attributes[2]);
    setPassword(attributes[3]);
    lock.unlock();
}
```

IV. Durability

This property guarantees that once a transaction has been committed, it will remain committed even in the case of a system failure (e.g., power outage or crash). This usually means that completed transactions (or their effects) are recorded in non-volatile memory.

As for achieving this property, my system is using the cache-aside strategy, which exports the result of any modifying operation result to the database with keeping a copy of the result in the cache. So, my system doesn't need it in my case.

If you're wondering how this property can be done. It can be done by exporting the operations done in the memory to a log file on the hard disk drive as a snapshot. Then if any systematic failure happens are any crash or so, the data that was in the volatile memory can be retrieved by executing all of the operations stored on that log file. Of course, storing on the log file needs some optimization, since there can be multiple operations done on the same attribute of the record, and what we care about is the last state of that record. So, it makes no sense to store all of the operations when we care about the last operation affected a particular record.

System's Architecture

The system is a java web app using maven. It uses the mvc pattern for its modules, which are designed using Servlets & Jsps.

I. (MVC): Model View Controller

Designing web apps was used to be simple. You just create the structure of the website using html, then styling it using css. Also, you can add some functionality to it using java script.

This was true, until we have started to use dynamic websites. Dynamic websites are usually connected to a database, which store all of its dynamic data. This design was complex to maintain and a lot of bugs came from no where and programmers spent hours in debugging their systems to determine the defected part.

All of that head ache was reduced by far when we start to use patterns for web apps. One of the most well-known patterns is called "MVC pattern".

MVC is the acronym of **Model View Controller**. Each of these words represent a module. Controller is the orchestrator between the model and the view. It is responsible for taking requests from the client side and sending back him back a response. Whereas the model is used to make queries from the database after connecting to it. The View module offers some web pages that can be dynamic to show the user his requested data.

So, client sends an http request, then the controller receive it and tell the model about the needed data. The model retrieves the needed data from the database and send it back to the controller. Finally, the controller sends the data, which is received from the model, to the view. The view creates the website with the passed dynamic data and the client see it.

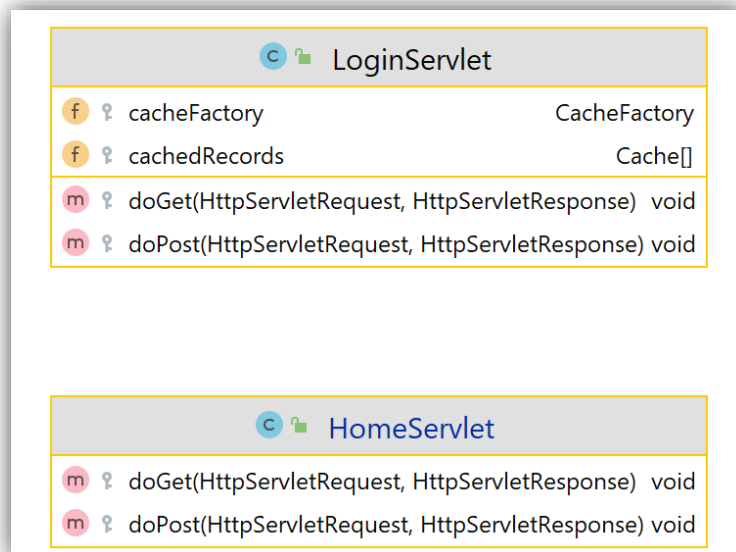
Have you noticed the simplicity factor of this pattern yet? The reason behind that simplicity was because of the separation of the business logic and needs module. Each module is responsible for a single task; hence we can track down the defect part by knowing the module responsible of.

As for my system, I've used the mvc pattern. The controller is designed using servlets, which is a technology used in java. As for the model, I've designed my own multi-threaded in-memory database and create the tables as well as the mapping classes. As for the view module, I've used

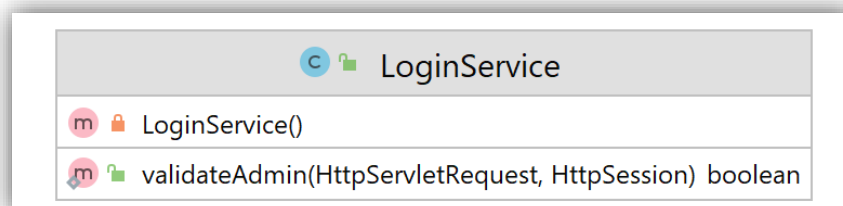
java server pages (also known as jsp) to create dynamic web pages. Jsp is more or less an html file with the support of writing java syntax inside it.

II. UML Class Diagrams

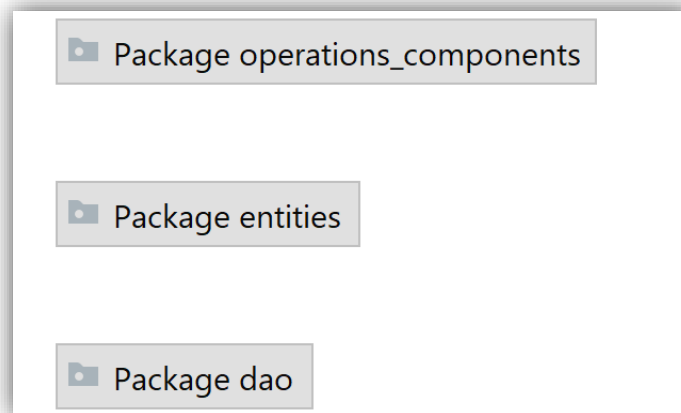
The following diagram depicts the controllers package, which consists of two controllers, one of them is used to manage logging the user in, and the other is responsible for redirecting him to the home page based on his authorization. In my system, there are two roles: Admin, who can do all of the CRUD operations. And the other role is the regular user, who can do the read operation only.



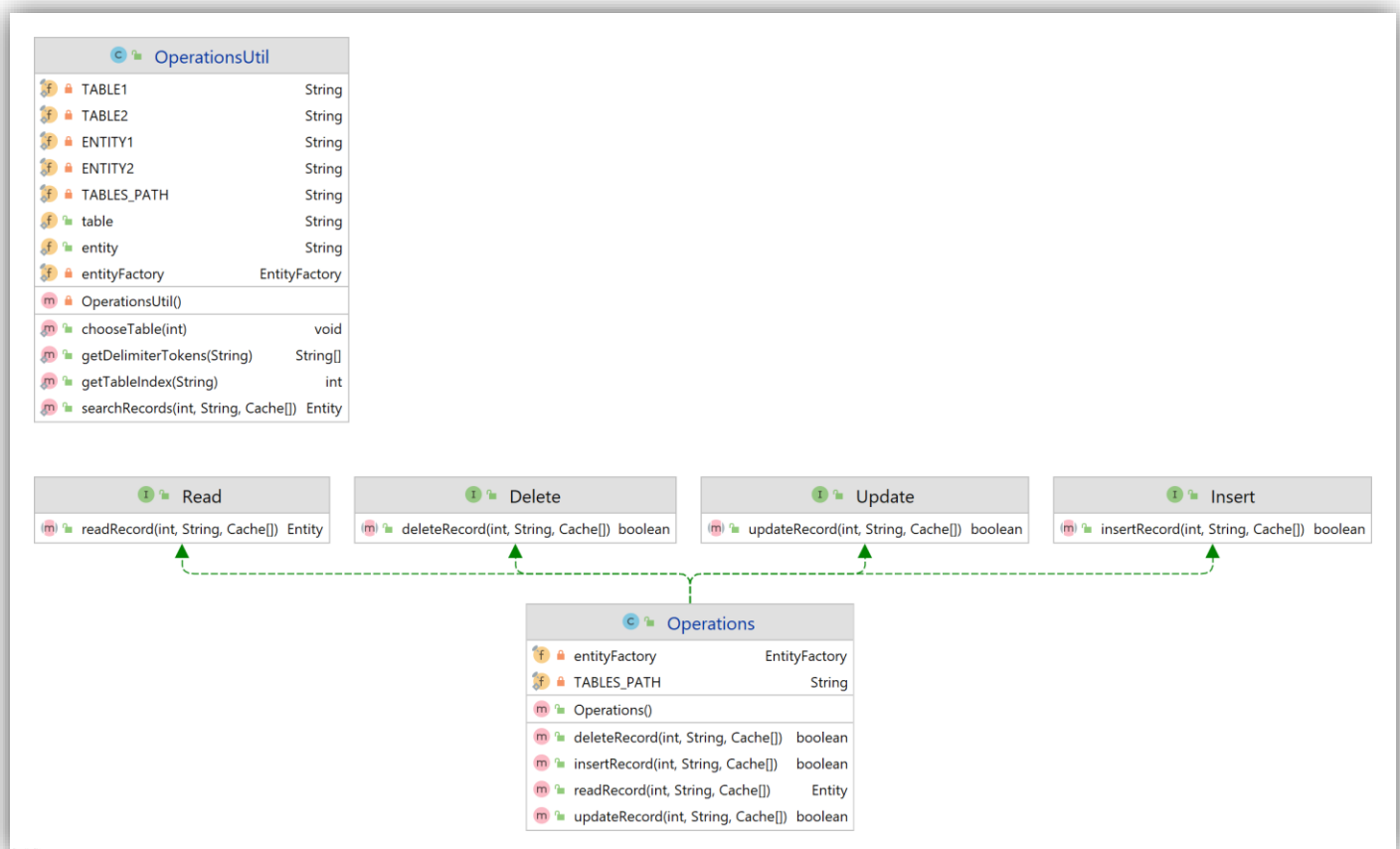
The following diagram depicts the services package, which consists of a single class for providing a login service to the login controller when the user attempts to login. In the login view, the user is asked to enter the admin password. If he fails to enter the right password, which is **0000**, he'll be redirected to the regular role view.



The following diagram depicts the model package, which consists of 3 packages, each of is responsible for a specific task.

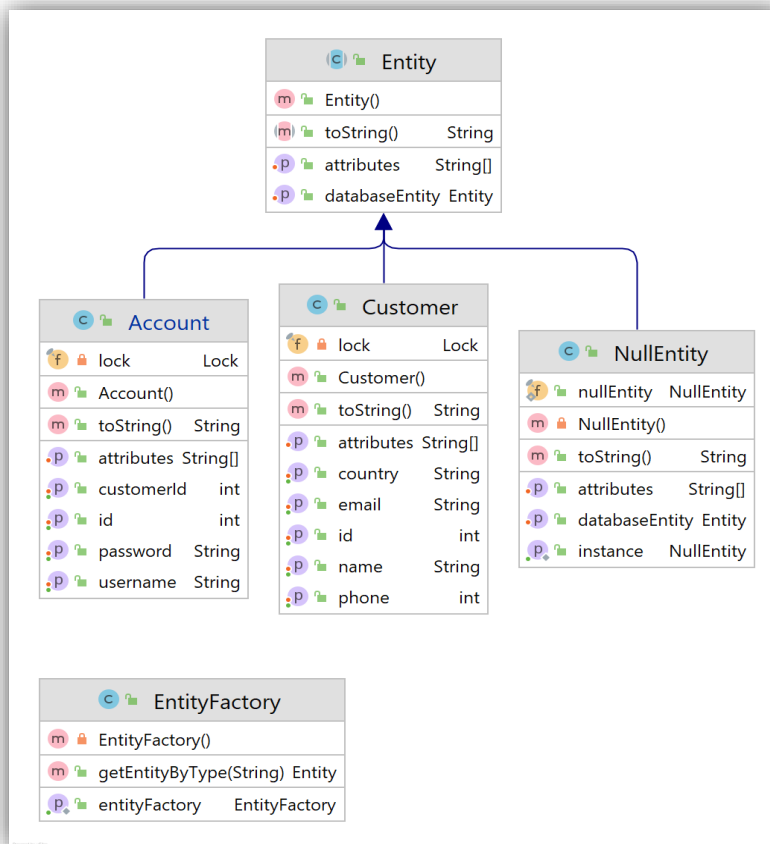


The following diagram represents the operations components package, which consists of the CRUD operation & a utility class that offers a searching service and other services.

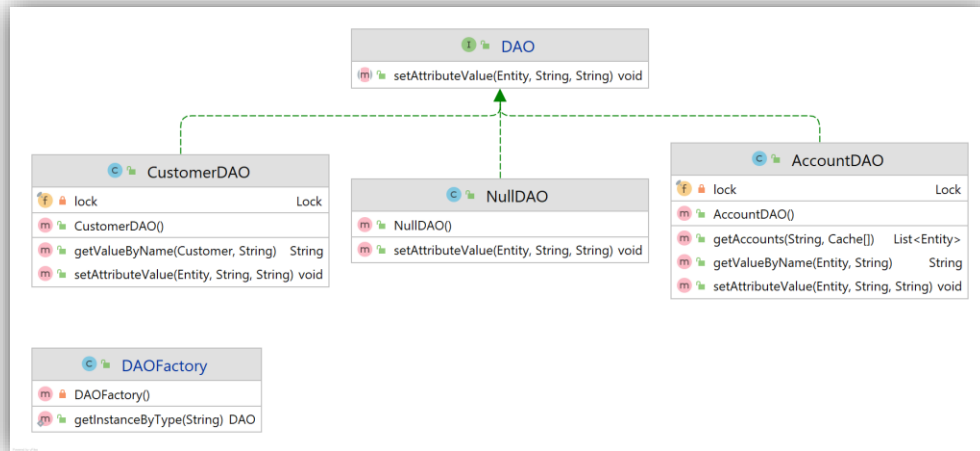


This diagram depicts entities package, which consists of the customer entity, the account entity and the null entity. Moreover, a factory method pattern is used inside the EntityFactory class.

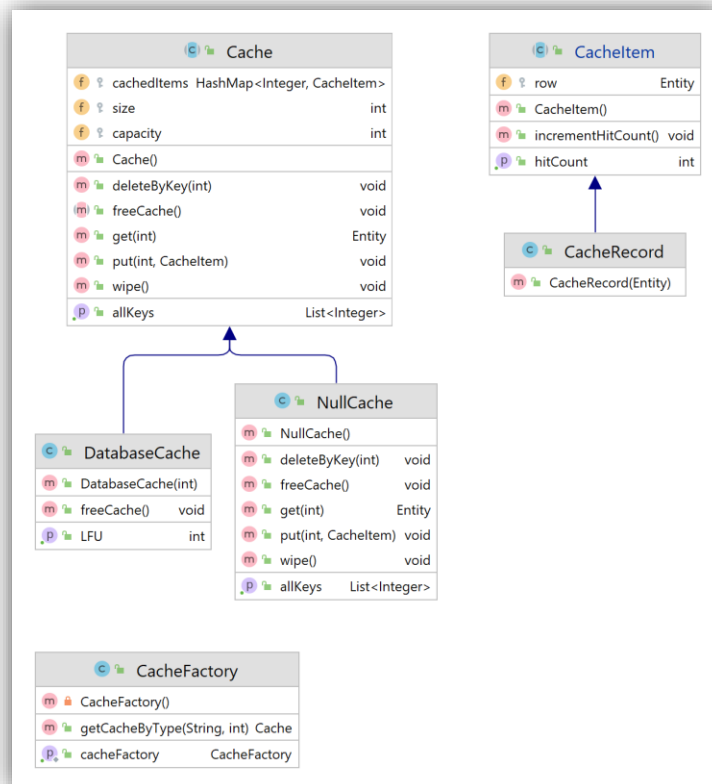
In my system, there is two tables. The first table is for customers of my system, and the other one is a table for accounts. Each user can have a single account, and each account can reference to a single account using a foreign key, so the relationship between both of them is 1-1.



This diagram depicts the data access object (DAO) package. It provides a data access object class for each entity. Also, it has a factory for getting an instance for the needed DAO.



This diagram depicts the cache components package. It consists of a **Cache** class, which is an abstraction layer. **DatabaseCache** and **NullCache** both implement it. Additionally, there is a **CacheItem** class and a **CacheRecord** class.



System's Data structures

I. Cache & Cache Item

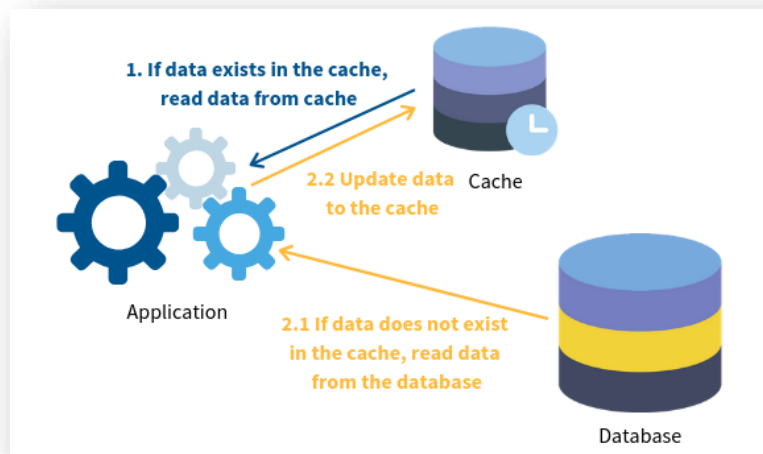
The cache is one of the most important and crucial things to think of when I designed the system. It should use proper data structures for doing the CRUD operations as fast as possible since the system is using an in-memory database. So nearly, all of the CRUD operations will be done on the memory with the help of the cache layer.

When I've thought of the cache, I need the fastest possible data structure for reading stored elements. I don't know any other data structure offers faster elements retrieval than a HashMap. My cache layer is basically a hash map, which stores the id of the records as a key, since it is a primary key and offers uniqueness. Whereas the value is CacheItem, which stores an entity with its hit count. An entity might be a customer record or an account record. Since there are two tables, I've divided the cache on both tables, so the final cache is an array of two DatabaseCache, each of is storing records of a particular table.

```
public abstract class Cache {  
    protected HashMap<Integer, CacheItem> cachedItems;  
    protected int size;  
    protected int capacity;  
  
    public void put(int key, CacheItem value) {  
        if (!cachedItems.containsKey(key)) {  
            if (size >= capacity)  
                freeCache();  
            cachedItems.put(key, value);  
            size++;  
        }  
    }  
}
```

```
public abstract class CacheItem {  
    protected Entity row;  
    private int hitCount = 0;  
    public int getHitCount() { return hitCount; }  
  
    public void incrementHitCount() { this.hitCount++; }  
}
```

I've gone with the cache-aside strategy. In this strategy, the cache is sitting aside the database. The application will first request the data from the cache. If the data exists (we call this a 'cache hit'), the app will retrieve the data directly. If not (we call this a 'cache miss'), the app will request data from the database and write it to the cache so that the data can be retrieved from the cache again next time.



When the cache gets full, eviction policy, I've gone with the Least Frequently Used (LFU) eviction policy, which depends on the hit count of the cache item. The cache item with the lowest hit count will be cleared out from the cache first. So, the hit count gets increased each time the system accesses a cache item.

```
@Override
public void freeCache() { cachedItems.remove(getLFU()); }

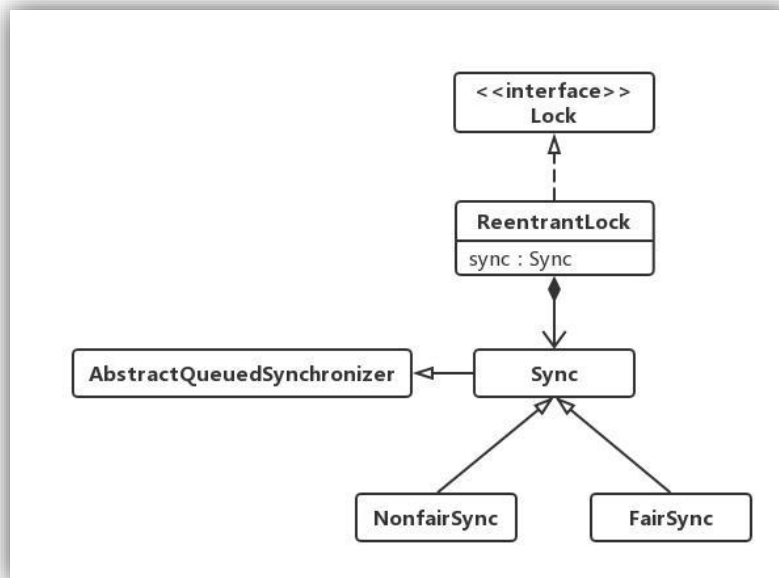
public int getLFU() {
    int minHitCount = (int) Double.POSITIVE_INFINITY;
    int minHitCountKey = -1;
    for (Map.Entry<Integer, CacheItem> entry : cachedItems.entrySet()) {
        CacheItem cachedItem = entry.getValue();
        if (cachedItem.getHitCount() < minHitCount) {
            minHitCount = cachedItem.getHitCount();
            minHitCountKey = entry.getKey();
        }
    }
    return minHitCountKey;
}
```

II. Reentrant Lock

The ReentrantLock class implements the Lock interface and provides synchronization to methods while accessing shared resources. The code, which manipulates the shared resource, is surrounded by calls to lock and unlock method. This gives a lock to the current working thread and blocks all other threads which are trying to take a lock on the shared resource.

As the name says, ReentrantLock allows threads to enter into the lock on a resource more than once. When the thread first enters into the lock, a hold count is set to one. Before unlocking the thread can re-enter into lock again and every time hold count is incremented by one. For every unlocks request, hold count is decremented by one and when hold count is 0, the resource is unlocked.

Reentrant Locks also offer a fairness parameter, by which the lock would abide by the order of the lock request. So, after a thread unlocks the resource, the lock would go to the thread which has been waiting for the longest time. This fairness mode is set up by passing true to the constructor of the lock.



Synchronization

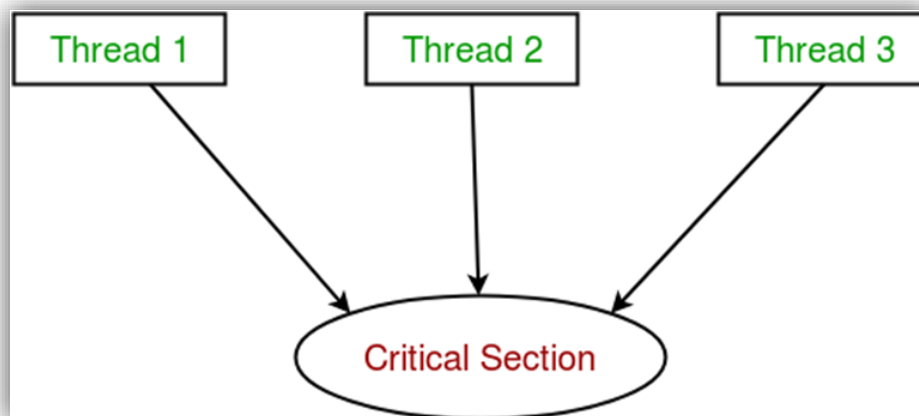
Concurrency refers to the ability to run multiple tasks at the same time, which can increase the efficiency of an application. Tasks can be submitted to run immediately, at a specified time, or in response to the completion of one or more actions upon which they depend. For example, an application that computes the total cost of a merchandise order must run a series of tasks. First, it looks up the state and local sales tax rates in one task while concurrently calculating the shipping costs in another. Then, in a third task that runs upon completion of the first two, it totals the results. Java SE and EE both support concurrency with a standardized API that enables the creation and scheduling of concurrent tasks.

Synchronization is all about ensuring that concurrent threads are changing the system between valid states.

Race condition is a concurrency problem that may occur inside a critical section. A critical section is a section of code that is executed by multiple threads and where the sequence of execution for the threads makes a difference in the result of the concurrent execution of the critical section.

When the result of multiple threads executing a critical section may differ depending on the sequence in which the threads execute, the critical section is said to contain a race condition. The term race condition stems from the metaphor that the threads are racing through the critical section, and that the result of that race impacts the result of executing the critical section.

Synchronizing the execution of the critical section is the solution here. Locks can be used for preventing more than a single thread of executing the critical section.



For the synchronization, my system always makes an object in the cache before doing the insert operation. Whereas it retrieves an object of the record from the cache if the case was cache hit or miss for the read, update or delete operations. This object will be the record that will be extracted to the cache and the database. After creating or retrieving the object, then it uses the set attributes method of the object which uses the setters to set the attributes of the record for doing the operation. The critical section for the system is the body of the set attributes method. I've used reentrant locks inside that method to prevent concurrent threads from accessing it.

```
@Override
public void setAttributes(String[] attributes) {
    lock.lock();
    setId(Integer.parseInt(attributes[0]));
    setCustomerId(Integer.parseInt(attributes[1]));
    setUsername(attributes[2]);
    setPassword(attributes[3]);
    lock.unlock();
}
```

```
@Override
public void setAttributes(String[] attributes) {
    lock.lock();
    setId(Integer.parseInt(attributes[0]));
    setName(attributes[1]);
    setCountry(attributes[2]);
    setPhone(Integer.parseInt(attributes[3]));
    setEmail(attributes[4]);
    lock.unlock();
}
```

CI/CD

I. CI: Continuous Integration

Continuous Integration is a concept where all of new features are continuously integrated in a software. Usually, there will be a pipeline of series of steps, which is performed on each new feature as soon as possible before adding it to an in-production software application.

The benefits of continuous integration and continuous deployment boil down to short feedback cycles. Before continuous integration, it was common for developers to work in isolated rooms for many weeks or even months without merging their work to the main line. Such a scenario usually results in problems when it was time to integrate. Their work would've diverged so drastically that insanely complex merge and logical conflicts would occur. That was when continuous integration was created. Each step in the CI/CD pipeline depends on the other. If any of the steps has failed, then the whole pipeline will be dropped. This shows how powerful and reliable the pipeline is. If the artifact, which is the final item to be deployed in java, is generated and has passed the pipeline successfully, then it is ready to be deployed to the production server, and by production server I mean that it'll be consumed by the clients.

The pipeline of CI often consists of the following two steps:

1. Build

The stage where the application is compiled. A building tool is often used for this step. In java, we have maven as well as gradle for the building. I've used gradle for the building step in my application.

2. Test

The stage where the result of the build is tested against bugs or failure. There are a lot of tests. But here we are interested in unit testing. It is done by programmers who write a lot of tests for particular units or blocks of code. Blocks of code can give a green light if they pass all of the tests, or a red light if they fail at least one of the tests. In java, we have Junit for doing unit testing. Unfortunately, I didn't do unit testing for my application. I didn't have the enough time to do it properly so I've skipped it. But usually, it is critical and important. The Testing step can save you your clients if a new infected block of code is prevented from being deployed to the production server.

II. CD: Continuous Delivery/Deployment

Continuous Delivery is the concept that follows the continuous integration pipeline. It comes to extend the CI pipeline further. While CI builds and tests a new added feature, CD helps delivering that tested feature to the production server. You might get confused about the difference between continuous delivery and continuous deployment. Do not do a mistake and think that they are the same because they are not. Often, there will be a step before the deploying to production step, which is deploying to staging. A staging area is where new features that are ready to be consumed are. Ready-to-use features usually stays in that staging area temporarily. When the feature is ready to be uploaded to the production server, then the new feature leaves the staging area to the next step using human intervention. This is the case for continuous delivery. For continuous deployment, it is nearly identical except for that the process that comes after of checking the new feature and the readiness of the production infrastructure is done automatically using automation tools. No human intervention is required for the feature to be deployed to the production servers after leaving the staging area, whereas in the continuous delivery, human intervention is required.

The pipeline of CD often consists of the following steps:

1. Acceptance Test

The step where the new feature is tested against the requirements, whether it is functional or non-functional requirements.

2. Deploy to Staging

In this step, the new feature is considered as a ready to consume by clients, but there are further checks done to check for the readiness of the production infrastructure and other elements.

3. Deploy to Production

In this step, the new feature is uploaded to the production servers. Hence it will be shown to the clients and they can start using and consuming it.

Finally, CI/CD both forms a pipeline that looks like the following:

Build => Test => Acceptance Test => Deploy to Staging => Deploy to Production.

Jenkins

It is a CI server that manages and control processes as code, build, test, deploy. Moreover, it can be used in DevOps environments to operate and monitor the environment.

For Jenkins to be run after installing it, java development kit (JDK), maven and tomcat are required to be installed on your machine. Then you can add a new pipeline after adding proper plugin to Jenkins. Finally, Jenkins should be allowed to upload and write to the tomcat server as it basically rebuilds your app after specifying a particular trigger with a schedule, and deploy that build after testing and packaging it to the tomcat server. So, it's preferred that you give Jenkins the permission to write on tomcat.

All

+

S	W	Name ↓	Last Success	Last Failure	Last Duration	
		First_Jenkins	1 day 3 hr - #2	N/A	1.2 sec	
		First_Web_App	1 day 1 hr - #3	N/A	13 sec	

Icon: S M L

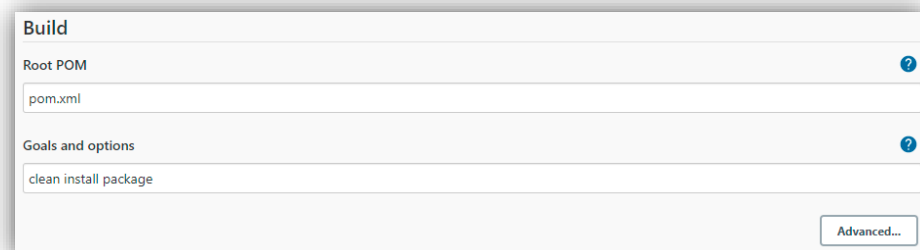
Legend

Atom feed for all

Atom feed for failures

Atom feed for just latest builds

Here is a screenshot of two items successfully deployed on Jenkins after setting up the previous steps. After uploading the project on a version source control system as github, just connect it to your item on Jenkins. I've specified the goals of the build step as clean install package, which are the goals for maven to do.



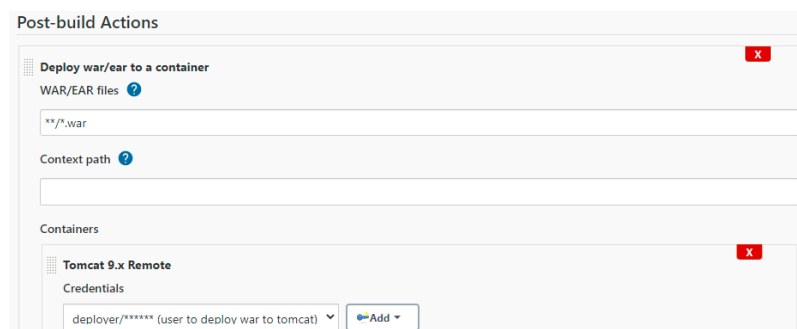
Build

Root POM

Goals and options

[Advanced...](#)

After doing previous steps, specify the type of the file which will be deployed to tomcat. It is a java web archive (war) in my case.



Post-build Actions

Deploy war/ear to a container

WAR/EAR files

Context path

Containers

Tomcat 9.x Remote

Credentials

[Add](#)

Finally, specify the schedule of the rebuild after the trigger is being caught, I've specified it with 5 stars

(* * * * *) each of represents a time segment (Year, Month, Day, Hour, Minute). And that's it. All of your modifications will be built, tested and deployed automatically after committing it in the version source control system.