Database

Table of contents

Omar Aljarrah

Breif Look

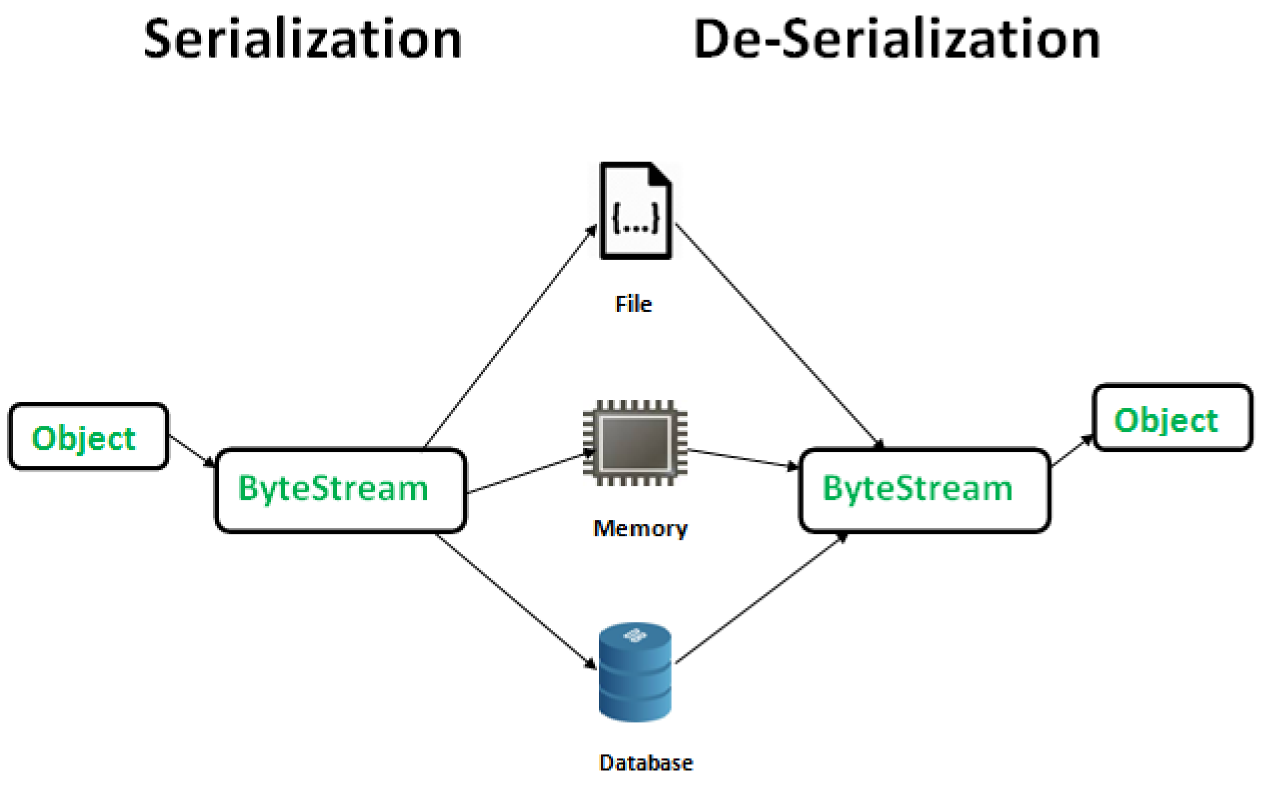
What does this project include?

- Writing, reading & logging mechanism.
- Multi-threaded database with 1 entity possible `Person` that has 2 properties `Name & Age` .
- Server that accepts requests from clients.
- Access management system to grant or denie access requests.
- Caching system to improve performance on read only operations.
- Concurrency management system to avoid inconsistency in the database.
- An interface that the user can use to interact with the database.
- Sticking to google java style.
- Sticking to the rules of system architecture design / SOLID principles (Allah is the only perfection in this world of imperfections).

Data Handeling

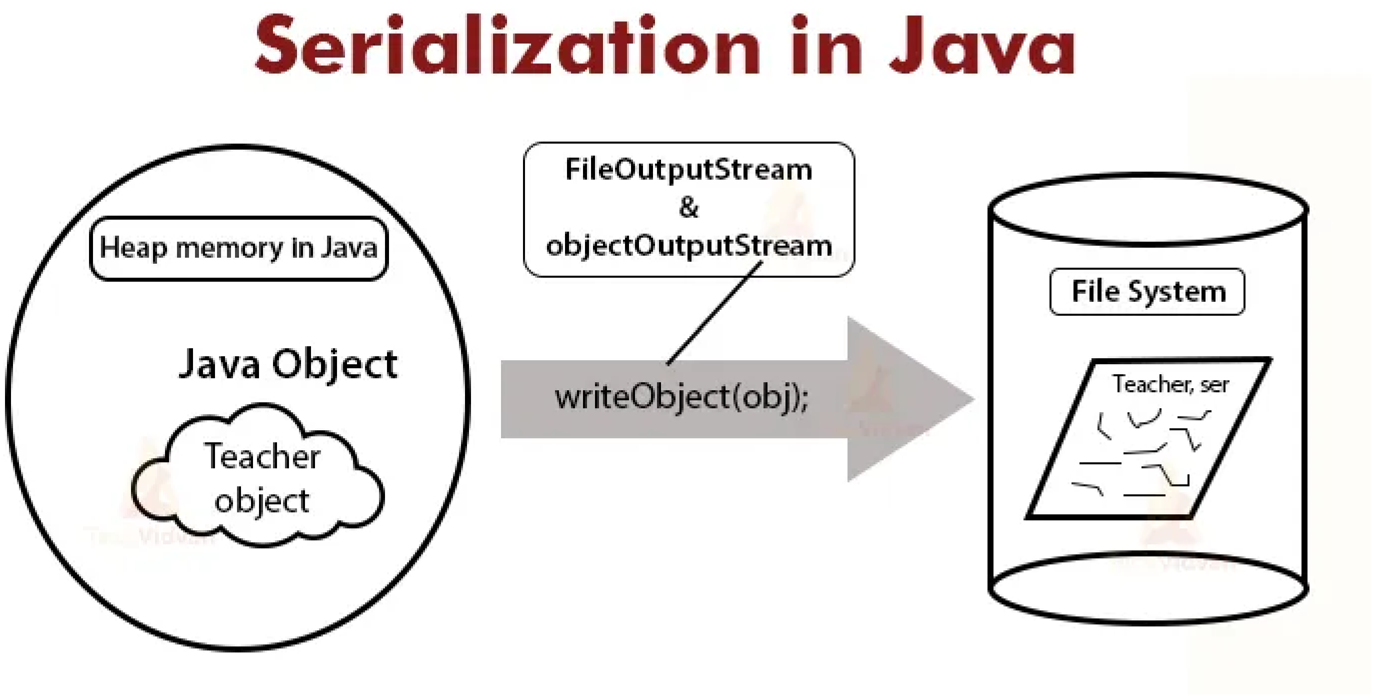
1) Serialization

- Serialization is the process of translating a data structure or object state into a format that can be stored (for example, in a file or memory data buffer) or transmitted (for example, over a computer network) and reconstructed later (possibly in a different computer environment). When the resulting series of bits is reread according to the serialization format, it can be used to create a semantically identical clone of the original object.
- For many complex objects, such as those that make extensive use of references, this process is not straightforward. Serialization of object-oriented objects does not include any of their associated methods with which they were previously linked.
- This process of serializing an object is also called marshalling an object in some situations. The opposite operation, extracting a data structure from a series of bytes, is deserialization, (also called unserialization or unmarshalling).
- All data that are to be stored on disk are written as serialized objects on text files, this mechanism can be achieved by implementing the `java.io.Serializable` & the `java.io.Externalizable` interfaces in these objects classes, this implies that retrieving data requires deserialization.



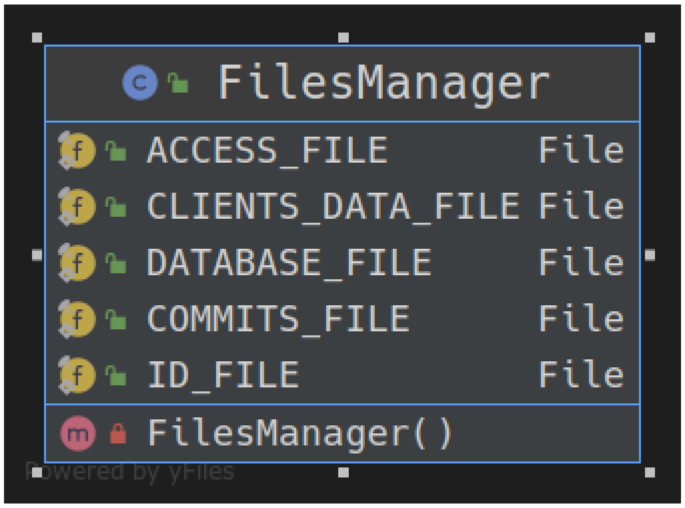
- Writing/Reading data is done using each of `java.io.ObjectInputStream` & `java.io.ObjectOutputStream` classes, given that each object that is to be read/written implements the serializable interface & has a special `serialVersionUID`.

- Requests/Responeses are sent through sockets as serialized objects, which makes compressing/decompressing data much easier, in addition to giving a general communication protocol that does not change through time if other classes are changed, though data has to be rewritten in some cases.



2) Files management

- All data are written as serialized objects on text files, files are stored as `static objects` in the `FilesManager` class, giving a reference to all the files this way provides more readability, and more independency to the classes, which implies that any future modifications on the files can be done easier without the need to modify code.



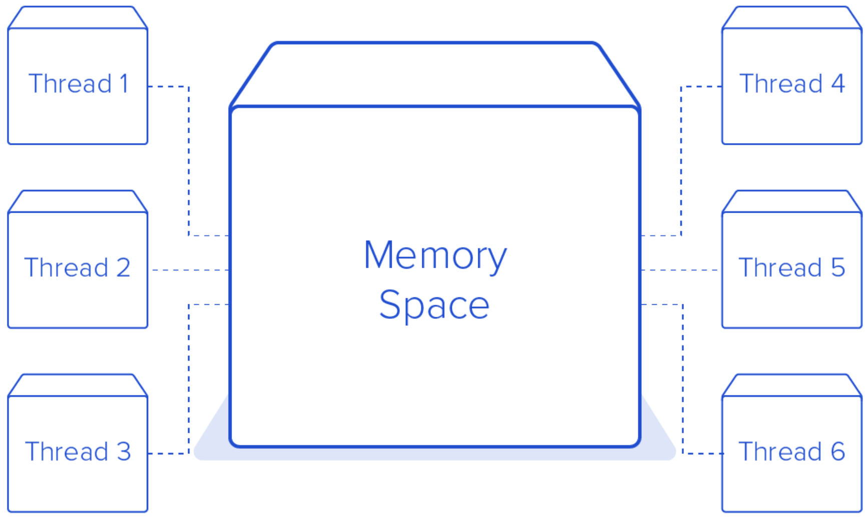
3) Reading/Writing & Memory

- All data that are to be written/read by the client do not use buffering, which insures the management of RAM usage, on the other hand, a cache is created for each client to increase performance of read operations.
- In order to clean the code, getting rid of the `try - catch` blocks & repeated code, I have made a tool to handle writing/reading files, providing the needed features for this specific task, and by creating interfaces as high abstraction, this serves good with the `SOLID principles` .

Concurrency

What is concurrency?

- concurrency is the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or at the same time simultaneously partial order, without affecting the final outcome. This allows for parallel execution of the concurrent units, which can significantly improve overall speed of the execution in multi-processor and multi-core systems. In more technical terms, concurrency refers to the decomposability of a program, algorithm, or problem into order-independent or partially-ordered components or units of computation.



Why concurrency is required?

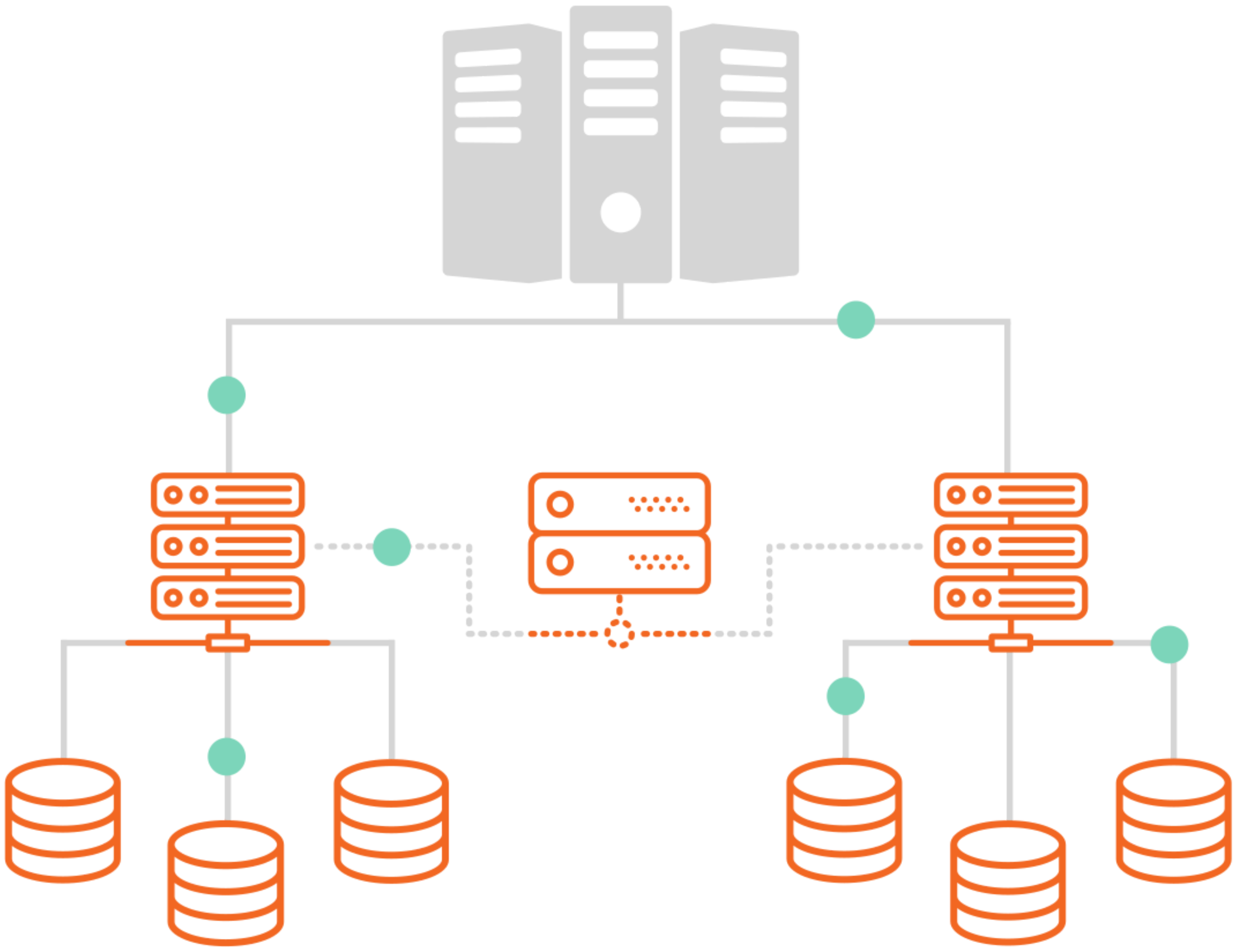
- Multithreaded applications can take full advantage of multiple processors to gain better performance through simultaneous execution of tasks. A well-implemented multithreaded application efficiently uses all the processors available for its own tasks where a single-threaded application must wait for each task to complete before continuing with the rest of the application. At no time can a single-threaded application execute on more than one processor in the system. In our case we need to process several requests as fast as possible in order to serve all clients, which makes a single-threaded algorithm an inefficient method to run anything similar to a database server.

How did we implement concurrency?

- We decided to go with a multi version concurrency control (`MVCC`) system, where each client that is to be served get's his own version of the database stored on the disk as a safe or private area, each version serves one client, then they can do whatever they want in that safe area of theirs, while we let the CPU handles each request as a separate thread, once the client is done they have to commit their modifications to the original database by sending a commit request.

```
databaseVersion = new Database(CoreDatabase.getNewVersion());
clientData = new ClientData(user, databaseVersion);
write(CLIENTS_FILE, clientData);
```

- As each client have their own version of the database, any modifications or reads will be processed using their version, which implies a sequential access to version per client.

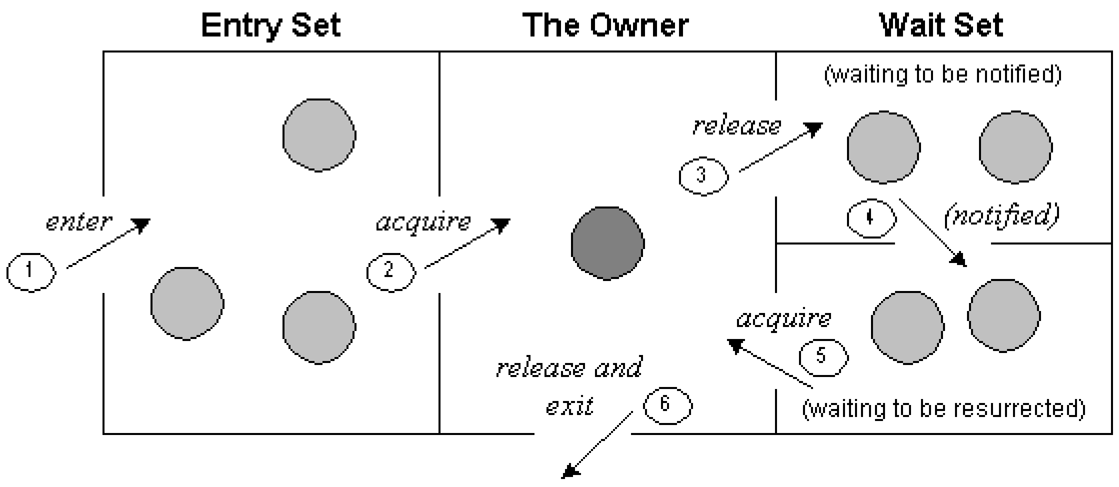


- For each request (CRUD operation) by the client, a new thread is created to process that reqest, and in order to assure consistency in data modifications, and to make sure that requests by the same client are proccessed in order, a lock on the data is created with each request, once the request is proccessed, the lock will be released & the data will become available again, this lock won't affect any other user as it only locks data that are present in one version (the version mapped to the client doing modifications), which means that other clients can access their data whenever they want, new clients can have thier new versions created at any time, though this availability comes at price (Increased disk usage), but for the commit operation, well that's another story.

```
synchronized (database.getDatabaseFile()) {
  --- Sorry but you can't access the data unless I say so :7 ---

  ~ A lot of code here ~
  ~ Proccess some request somehow ~

}
--- Done, now you can access the data again ^^ ---
```



- As we said, commits are different story when it comes to synchronization, so what about them?

How the commit works?

- A user can do commit once per session, in a sense it is the same as any CRUD operation, but it locks both original & client's versions, which means that while a commit operation is on, no other client can access the the original data, in other words, new client's requesting a version of the database have to wait until the ongoing commit is done, also any other client requesting to commit have to wait too, this method assures consistency in data, but still, a problem comes up, which is the conflict that comes out of different versions, this problem can be solved by few methods, the one we picked theoretically works fine, but it has few issues too, which is aborting commits that can not be applied.

- Assuming past commits, each successful `commit` operation has 2 data sets, `readSet` that contains the `IDs` for all the records that are read by the client, & a `writeSet` that contains the

`IDs` of any modification on the database in that commited version.

- Once a client send a `commit` request, it will check if the commit operation is valid or not, and that is done by comparing the current commit request with the past commits, the checker will only compare the current commit to those commits that have been done at the time gap in between the current client first connection moment (that moment when their session started) & the commit moment (current moment). Comparison is done by comparing the current commit `readSet` & `writeSet` with other commits `writeSet` , if there is a conflict the commit will abort and throw an `InvalidCommitException` , otherwise it will be accepted and the data will get written to the original database, and a response of the commit status will be sent as a response to the client.

```
function checkCommitStatus() {
  for (commit : commitsList) {
    clientCommitDate = commit.getCommitTime();
    data = commit.getTransactionData();

    timeConflictExist = clientCommitDate.after(clientData.getConnectionTime());
    dataConflictExist = transactionData.checkConflict(data);

    if (dataConflictExist && timeConflictExist) {
      throw new InvalidCommitException();
    }
  }
}

-----------------------SOME CODE-----------------------

try {
    checkCommitStatus();
    registerCommit();
    CoreDatabase.commit();
    return new AcceptedCommit();
  } catch (InvalidCommitException invalidCommitException) {
    return new AbortedCommit();
  }
```

- If the current `commit` is accepted, it will be registered as a past commit using the function `registerCommit` .

```
writer.write(commitsFile, this);
```

- Once the client requested the commit operation, either their commit got accepted or aborted, their session will be cleaned, and their version of the database will be discarded.

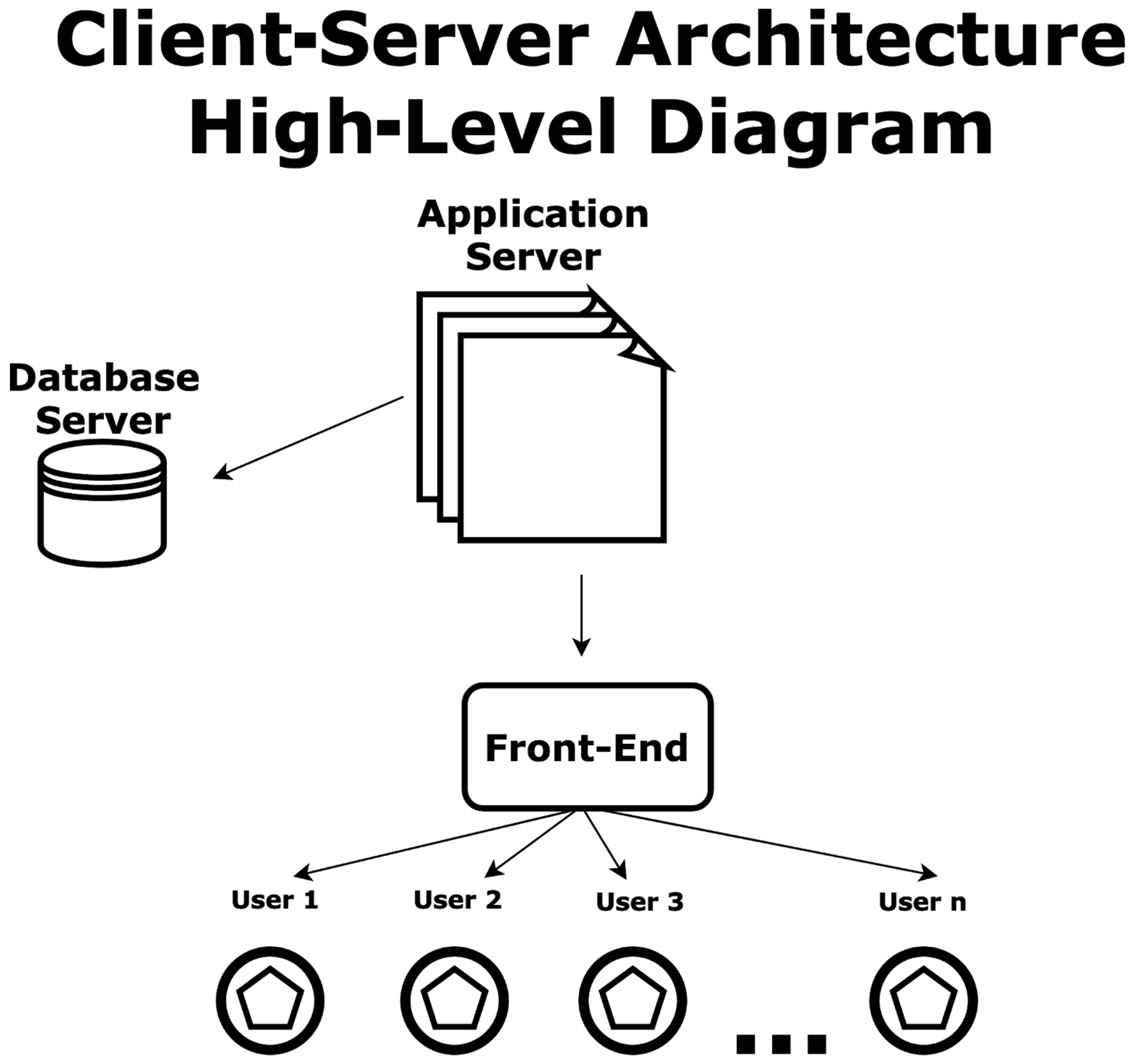
- It's worth to mention that the commit operation will lock both the original & client version of the database in a synchronized enviroment, that is the only moment when the original database is being locked.

- This method has the advantage of almost not using locks on the original database except at the commit stage which implies that the performance should be maximized, but the cost for that extra performance is the huge extra usage of the disk memory, but as disk memory isn't that big issue now a days, it can be more convenient to use the `MVCC` system, but that does not imply that this solution has no issues at all.

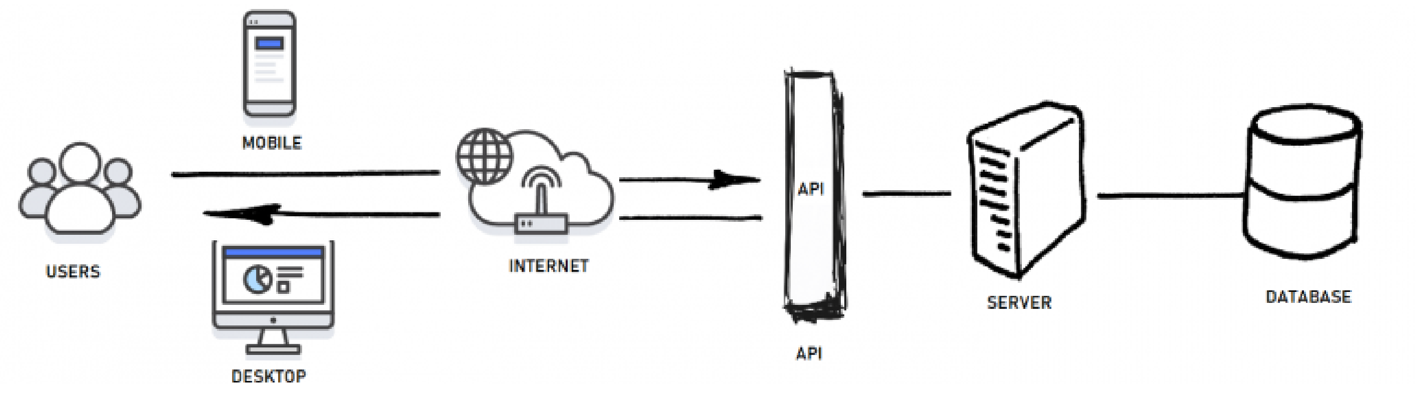
Connection

Connection

- Connection describes a process in which two or more computers or devices transfer data, instructions, and information. Some communications involve cables and wires; others are sent wirelessly through the air.
- In our project, we have applied a `Client - Server` model, that is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients.
- Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs, which share their resources with clients. A client usually does not share any of its resources, but it requests content or service from a server. Clients, therefore, initiate communication sessions with servers, which await incoming requests.
- The "client-server" characteristic describes the relationship of cooperating programs in an application. The server component provides a function or service to one or many clients, which initiate requests for such services. Servers are classified by the services they provide. For example, a web server serves web pages and a file server serves computer files. A shared resource may be any of the server computer's software and electronic components, from programs and data to processors and storage devices. The sharing of resources of a server constitutes a service.

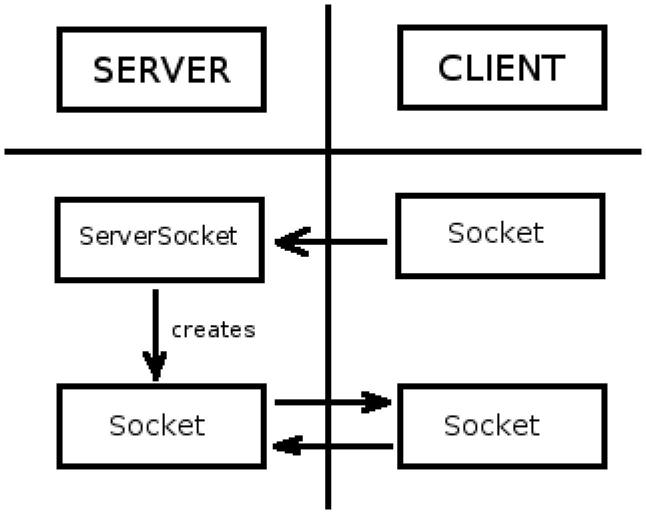


- Clients and servers exchange messages in a request–response messaging pattern. The client sends a request, and the server returns a response. This exchange of messages is an example of inter-process communication. To communicate, the computers must have a common language, and they must follow rules so that both the client and the server know what to expect. The language and rules of communication are defined in a communications protocol. All protocols operate in the application layer. The application layer protocol defines the basic patterns of the dialogue. To formalize the data exchange even further, the server may implement an application programming interface (API).



What did we use?

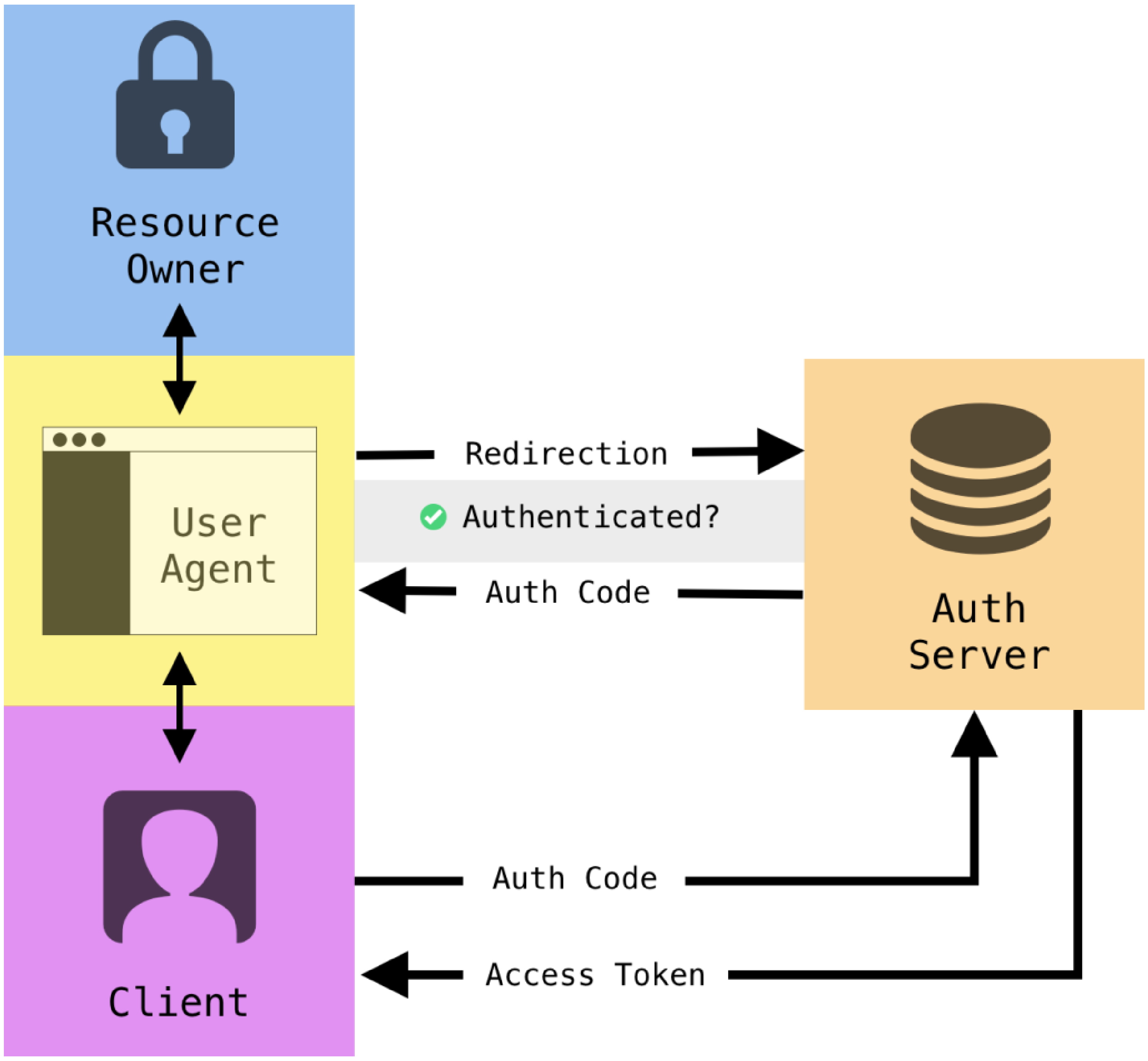
- We use Sockets to create a `Client - Server` enviroment, where both client & server use their sockets to communicate with each other, a server has a port in which it listens to clients requests on it, a client sends a request to that port, then they are connected, this mechanizm is provided by both the `ServerSocket` for the server side, `Socket` for the client side, in which both tools are provided by the `java.net` API.
- A `ServerSocket` job is to listen to requests, while in order to send & recieve data with other sockets, it creates a `Socket` object automatically once a request is accepted, that is configures to communicate with the socket requesting connection. The actual work of the server socket is performed by an instance of the SocketImpl class. An application can change the socket factory that creates the socket implementation to configure itself to create sockets appropriate to the local firewall.



Client Side

- In order for the client to establish connection with the server, they will send a connection request of type `ConnectionRequest` that should be which in place will configure ask for authorization from the access layer by sending a `LoginRequest` .

- Once they get access, they will open a new socket for connectin with the server, then the client is registered on the server as an ongoing connection.

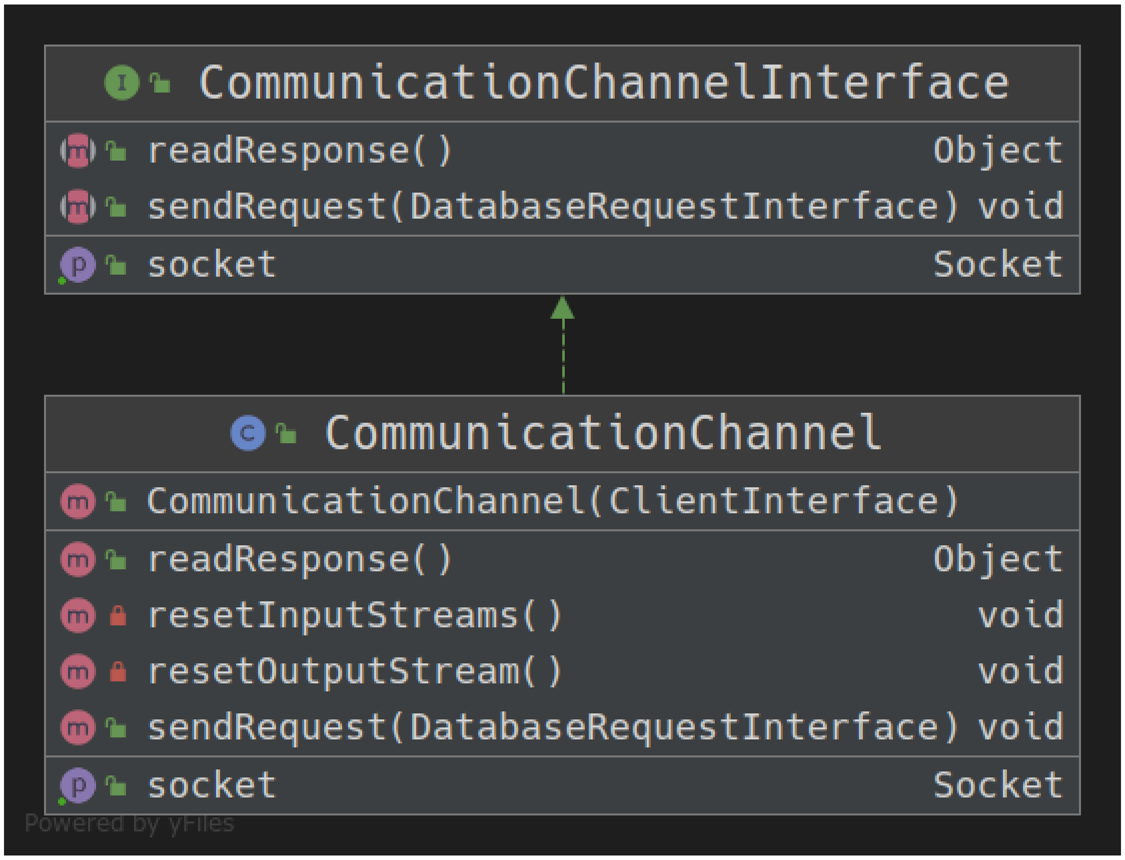


- Once a user is connected to the server, in order to do any of the CRUD operations he has to use the provided interface `DatabaseApi` , that for each CRUD operation, there will be a new request of type `DatabaseRequest` that includes the data of the requested operation sent to the server, each request opens a new socket at the client side while the old socket (if any exist) is closed.

```
function initConnection() {
  if (clientConnection != null) {
    clientConnection.getSocket().close();
  }
```

```
    clientConnection = new ClientConnection(this);
  }
```

- Each request sent or response recieved at the client side is handled using the `RequestResponseHandler` , that is an interface (as a concept) for using a special channel of type `CommunicationChannel` , the communication channel handles creating sockets, sending new requests & reading server responses.



- Any CRUD operation made through the API, will make a new object of type `Query` that will generate the right action request, then send it through the communication channel to the server, the `Query` class works as backend for the `DatabaseApi` .

Server Side

- Once the client pass the routing layer & the access check, each query he sends is captured by the server in the main loop, and the request gets read inorder to be processed.

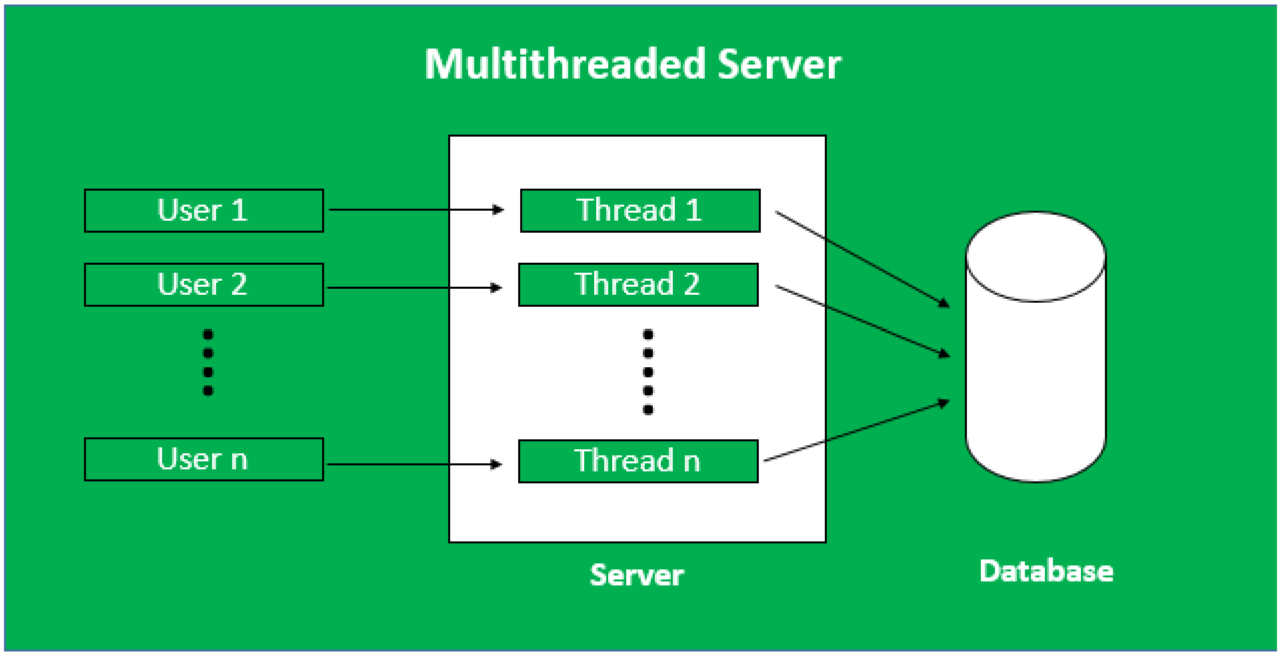
```
Main Loop {
  socket = serverSocket.accept();
  reader = inputStream(socket.getInputStream());
  request = inputStream.readObject();
}
```

- After the request is read & is ready to be processed, the server will start a new thread to process the request as each request is processed independently, then the server lets the `ClientHandler` class process the request.

```
---request got read---
new Thread(
          new ClientHandler(request, socket)
  ).start();
```

- The `ClientHandler` will ask for the client version of the database, that is stored on the disk & mapped to that specific client, as each client gets a version once they start a connection, the client handler will proccess the reqests on that version in a separate thread.
- In case a response by the server is expected, the `ClientHandler` class will handle sending the response through the server socket.

```
if (responseExpected) {
      outputStream = OutputStream(socket.getOutputStream());
      ---Processed the request---
      outputStream.writeObject(response);
 }
```



- Unlike clients where there might be more than one client, a server is a server, in other words there is only one server to serve all clients, this implies that using a design pattern like singelton a sensible decision to make.

Access Management

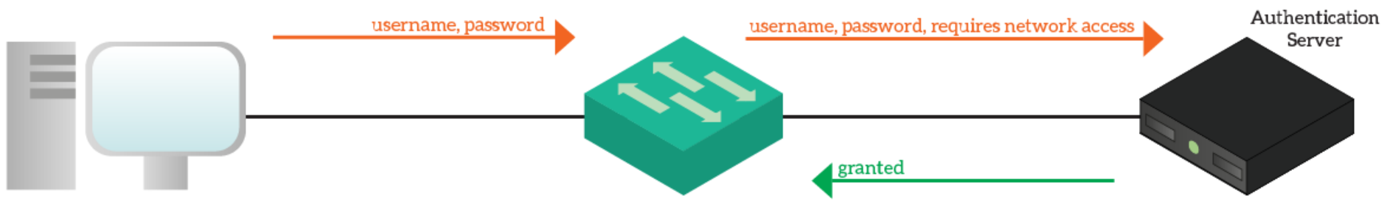
Users

- There are 2 types of users:
    - Normal User: their role is to make a connection to the server & do the CRUD operations on the database if and only if they are valid registered users.
    - Admin User: their role is to give access for normal users so they can do their role, also they can turn their access down by deleting them from the list of registered users.

Grant Access or Denie it

- Once a client tryes to connect to the server, he goes through an access layer first, this layer checks if the user trying to connect has the right to do so, then it sends a respose to the user connection request wether they got access or not, in case the user does not have the authority to connect, it will send him a `DeniedAccess` object, then a denied access exception should be thrown at client side.

```
if (!access.getAccess()){
  throw new DeniedAccessException();
}
```



Access Database

- Users with access privilege are stored as serialized objects on a file `access.txt` , each user has 3 properties, `username` , `access type` & a special `ID` , in order to get access, a user should send a connection request to the routing layer, a connection request includes each of the `username` & `IP address` of the client.

```
function connect(USERNAME, IP) throws DeniedAccessException {
  request = new ConnectionRequest(USERNAME, IP);
}
```

- Once a user get's access he will be registered at the server side.

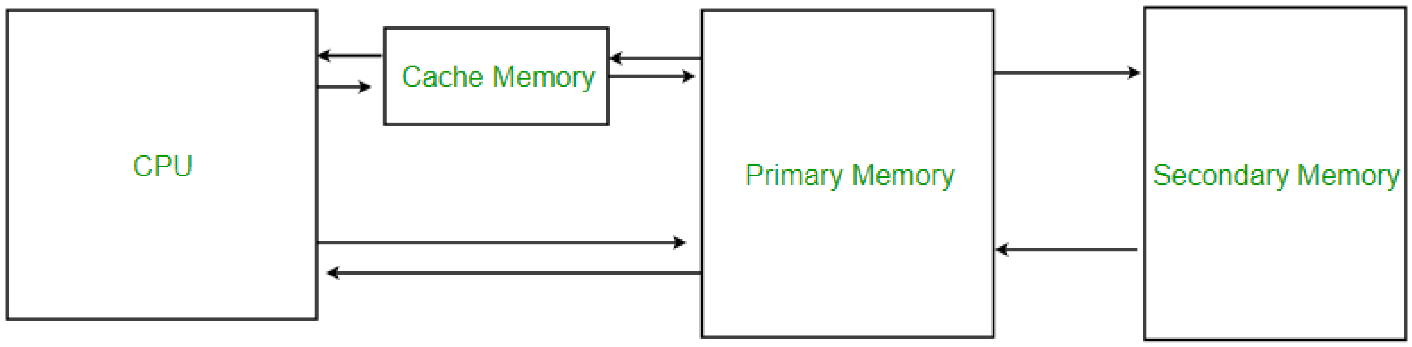
```
login = readLoginRequest();
access = ServerAccess.checkAccess(login);
if (access.getAccess()){
  registerClient(access, login);
}
```

- Registration process opens a new session for the client, giving him a version of the database that is present on the disk until the client finish their session.

Cache

What is cache?

- Cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache, which is faster than recomputing a result or reading from a slower data store; thus, the more requests that can be served from the cache, the faster the system performs.



Why use cache?

- The data in a cache is generally stored in fast access hardware such as RAM (Random-access memory) and may also be used in correlation with a software component. A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer. Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable. A database cache supplements your primary database by removing unnecessary pressure on it, typically in the form of frequently accessed read data. The cache itself can live in a number of areas including your database, application or as a standalone layer.

What is cached?

- Whenever a new client connects to the server and have their own version of the database, a new read-only cache is generated to improve performance, the generated cache will contain a random number of data records that does not exceed 10% of the total records count in the database.

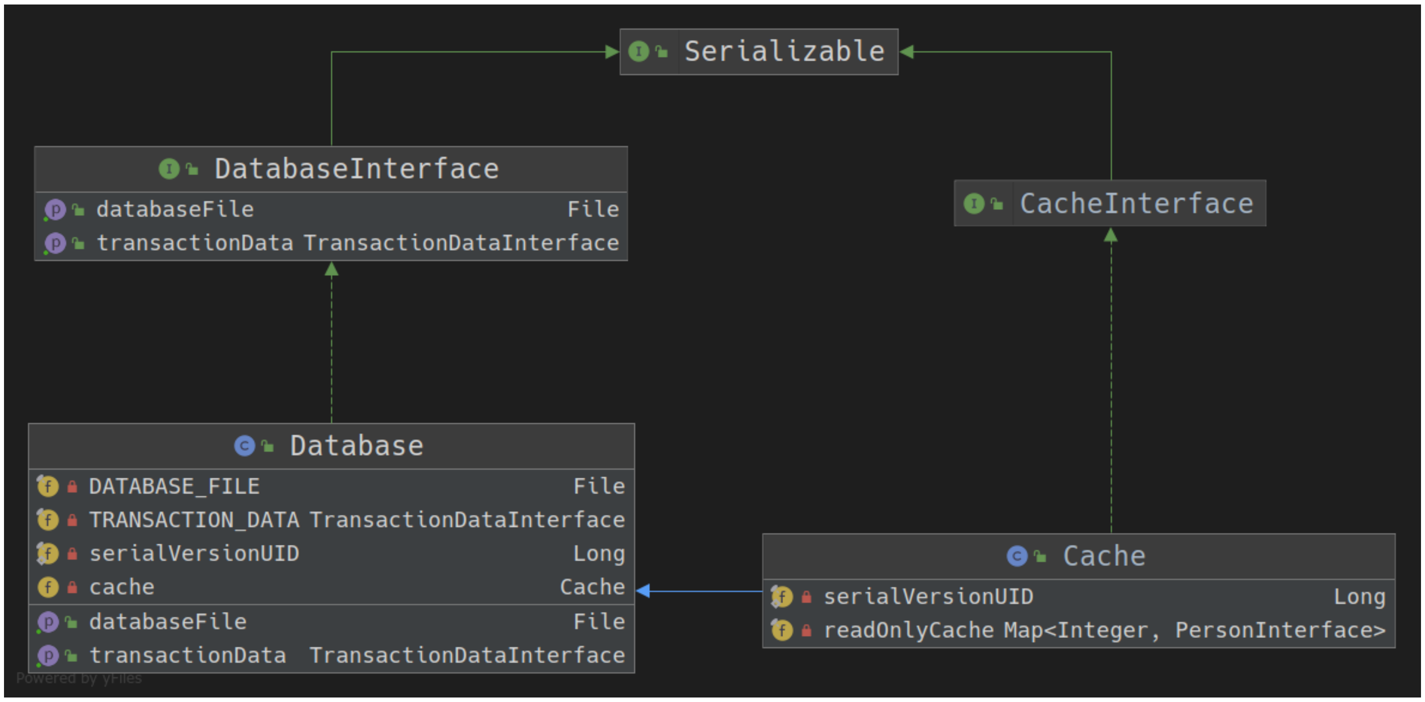
```
function generateRandomCacheSize() {
  min = 1;
```

```
    max = totalRecordsCount/10;
    pivot = (max-min+1)+min;
    return floor(Math.random() * pivot);
  }
```

How does cache work?

- Whenever a client attempts to read some entity, the server side application will try to get the record out of the cache, if the record does not exist in the cache, it will go & read data from the disk.

```
  try {
    cacheRead = cache.read(recordId);
    return (PersonInterface) cacheRead;
  } catch (NullPointerException nullPointerException) {
      return (Person) diskRead(recordID);
  }
```



- It is worth to say that any modifications on the data are not performed on the cache, this can be an issue for Some, but it's acceptable as long ass the cache is used for read only purposes.

More on the Workflow

- When a client imports the `DatabaseApi` , they create a new instance of the class passing their `username` & `ipAddress` as a parameter, creating a new instance will configure the connection with the server.

```
public DatabaseApi(String username, String userIp) throws IOException, DeniedAccessEx
  client = new Client(username, userIp);
  client.connect();
}
```

- The `connect()` method invoked by the client instance will create a new `ConnectionRequest` instance, then the `getDatabasePort(ip)` will handle requesting access back the scene.

```
public void connect() throws DeniedAccessException, IOException {
  ConnectionRequestInterface request = new ConnectionRequest(USERNAME, ip);
  this.socket = new Socket(ip, request.getDatabasePort(ip));
}
```

- Whenever the client invoke any of the `DatabaseApi` methods, it will create a new `Query` instance, this query instance will create a new `CommunicationChannel` that is responsible for sending reqests & reading responses.
- When the client passes both the routing layer & access identifier, his version of the database is created and a new thread will be created for each request, all their queries will be sent to the `QueryTranslator` by the `ClientHandler` , that is responsible for translating the request.

```
translator = new QueryTranslator(database);
translator.translate(request);
```

- Once the client is done with his queries, they will invoker the `commit` method, that will work as mentioned before.

SOLID Principles

I did what I could, may Allah help me :')

1) Single Responsibility Principle (SRP)

S/W engineer
Author
News paper editor
CAN'T YOU DO ANYTHING RIGHT?
Copyright 2003 Randy Glasbergen. www.glasbergen.com

- What is SRP? The Single Responsibility Principle is a computer programming principle that states that every module or class should have responsibility over a single part of the functionality that is provided by the software. That responsibility should be fully encapsulated by that class. A good quote relating to this principle is `A module should be responsible to one, and only one, actor. --uncle_bob`.
- Why is it important? The reason it is important to keep a class focused on a single concern is that it makes the class more robust. If the code needed to be changed constantly, there is a chance of the code becoming unstable and eventually leading to a break further down the development line.
- So... what did I do about it? well... I focused on giving each module a one & only one goal that it works to achieve, a `Log` class for logs, `Writing/Reading` tools as separated modules, splitting `CRUD` operations into a module per operation & a lot other than that are an example for of SRP in the code base.

2) Open Closed Principle (OCP)

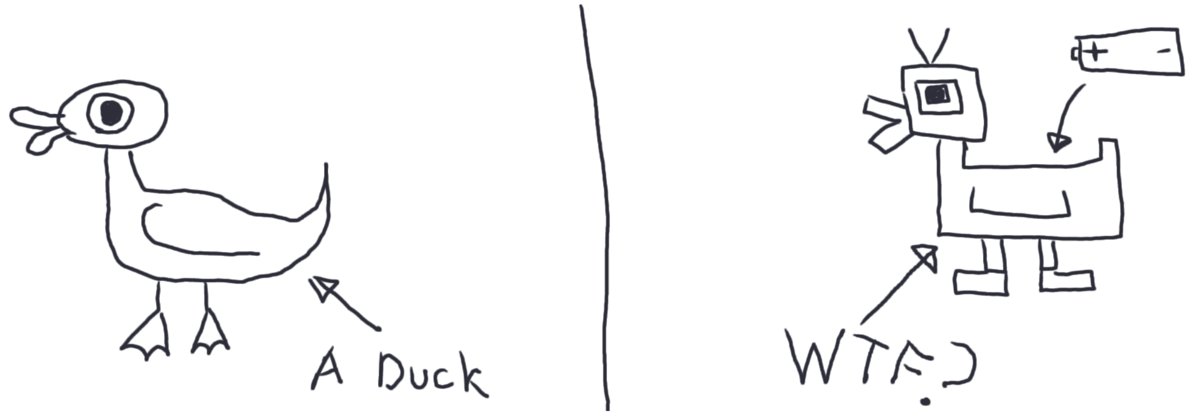
- What is OCP? The Open-Closed Principle (OCP) was coined in 1988 by Bertrand Meyer. It says:A software artifact should be open for extension but closed for modification.In other words, the behavior of a software artifact ought to be extendible, without having to modify that artifact. This, of course, is the most fundamental reason that we study software architecture. Clearly, if simple extensions to the requirements force massive changes to the software, then the architects of that software system have engaged in a spectacular failure.
- When a module is open? A module will be said to be open if it is still available for extension without the need for massive changes in other modules. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.
- When a module is closed? A module will be said to be closed of changing/adding any kind of extension implies doing massive changes on other modules, which can lead to instability & inconsistency in both the code & development cycle.
- What did I do here? I focused on using interfaces where the implementations can be changed and multiple implementations could be created and polymorphically substituted for each other. Interface specifications can be reused through inheritance but implementation need not be. The existing interface is closed to modifications and new implementations must, at a minimum, implement that interface which provides a protocol that derived classes implement.
- Why interfaces assures that OCP is safe? changes to derived classes which implement the interface will not break any client code, and may not even require recompilation of some clients. What we can’t do is change the interface definition. Any change here may force changes on most or all of its clients. Abstract interfaces directly support the Open/Closed Principle. They must be extended, but are closed to modification. Since they have no implementation they have no latent errors to fix and no performance issues.
- But is it really possible to stick `100%` to the OCP principle? Few would suggest that 100 percent of every design should satisfy the open/closed principle. But an effective design may use many components which do satisfy the principle but also include `program glue` . The `glue` is used to

bind the components into a working program, without much regard to the open/closedness of the `glue` part.

3) Liskov Substitution Principle (LSP)

- What is LSP? Substitutability is a principle in object-oriented programming stating that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., an object of type T may be substituted with any object of a subtype S) without altering any of the desirable properties of the program (correctness, task performed, etc.). More formally, the LSP is a particular definition of a subtyping relation, called (strong) behavioral subtyping. It is a semantic rather than merely syntactic relation, because it intends to guarantee semantic interoperability of types in a hierarchy, object types in particular. To ensure a design supports the LSP:
    - Derived objects must not expect users to obey pre-conditions stronger than expected for the base class.
    - Derived objects must satisfy all of the post-conditions satisfied by the base class.

  LSP can also be described as a counter-example of Duck Test: “If it looks like a duck, quacks like a duck, but needs batteries – you probably have the wrong abstraction”



- Why LSP is important? The Liskov Substitution Principle is the third of Robert C. Martin’s SOLID design principles. It extends the Open/Closed principle and enables you to replace objects of a parent class with objects of a subclass without breaking the application. This requires all subclasses to behave in the same way as the parent class.

- How did I implement it? As in OCP, interfaces assures the substitution of child classes, all classes that are childs of a higher abstract class (interface), the compiler itself won't let you forget the rules defined by the parent class, which implies that childs of the same interface are always substitutabile.

4) Interface Segregation Principle (ISP)

image

- What is ISP? (ISP) states that no client should be forced to depend on methods it does not use.[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called role interfaces.[2] ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

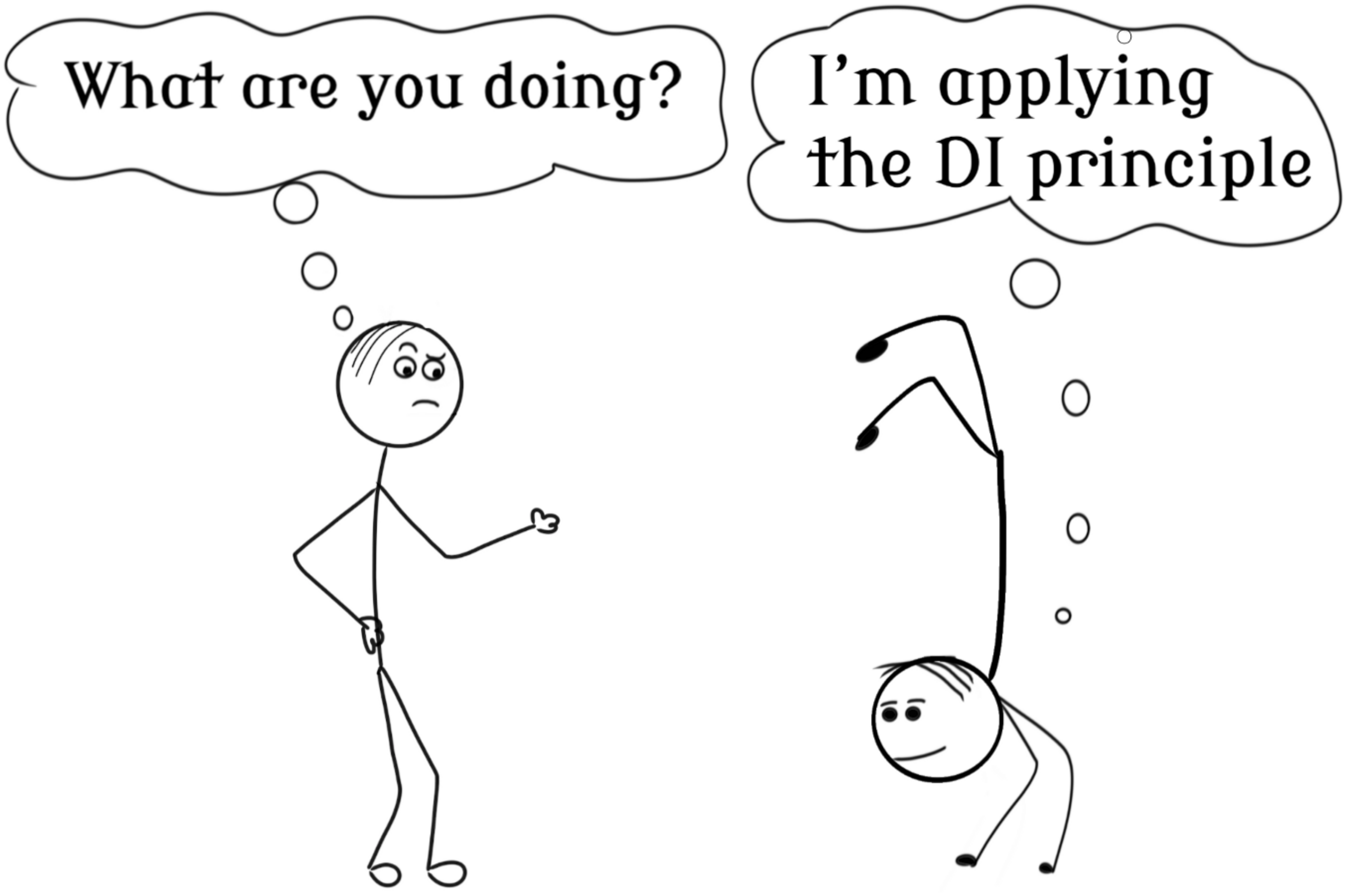
- means that sometimes we tend to make interfaces with a lot of methods, which can be good to an extent, however this can easily abused, and we can end up with classes that implement empty or useless methods which of course adds extra code and burden to our apps. Imagine you are declaring a lot of methods in single interface, if you like visual aids a class that is implementing an interface but that is really needing a couple of methods of it would look like this:

In the other hand, if you properly apply the interface segregation and split your interface in smaller subsets you can me sure to implement those that are only needed:

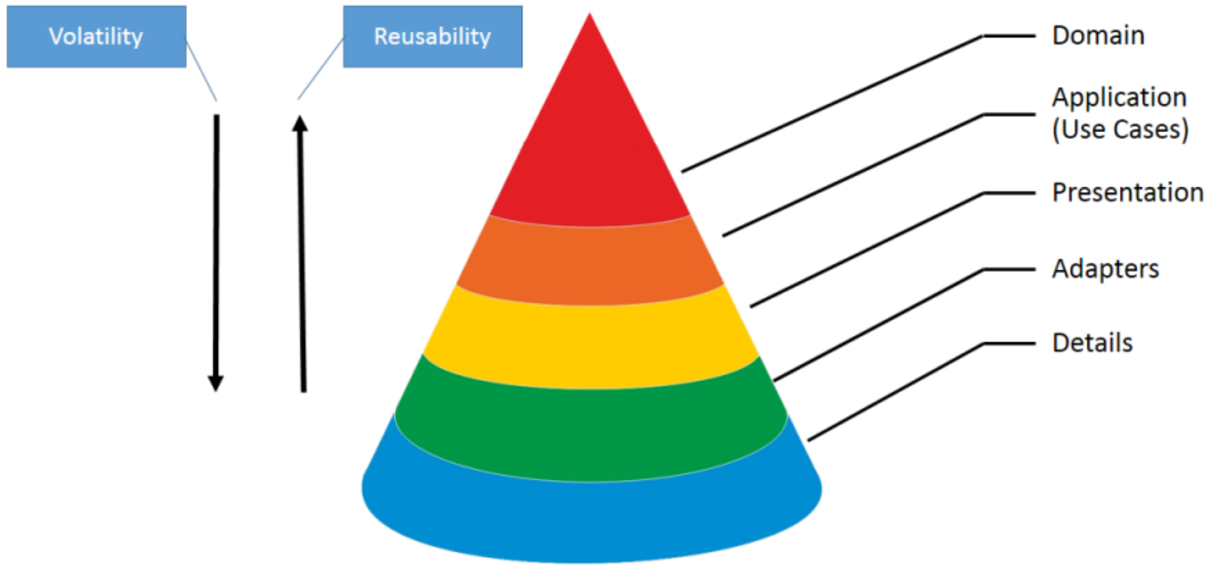
See! Is way better! Enforcing this principle will allow you to have low coupling which aids to a better maintainability and high resistance to changes. So you can really leverage the usage of interfaces and implementing the methods when you really should.

- How did I implement the ISP? Simply implemented the SRP on interfaces, an example of that is the reading & writing tools, as each tool of those two were splitted into different modules, their interface were splitted too, if you want to read only, why compile the write module? another example is the requests, as there are `DatabaseRequest, ConnectionRequest & LoginRequest` , instead of making them a child to one huge parent (interface) and violating the ISP, each of them has its own interface.

5) Dependency Inversion Principle (DIP)



- What is DIP? DIP is a specific form of loosely coupling software modules. When following this principle, the conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are reversed, thus rendering high-level modules independent of the low-level module implementation details. The principle states:
  - High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions. The idea behind the previous points of this principle is that when designing the interaction between a high-level module and a low-level one, the interaction should be thought of as an abstract interaction between them. This not only has implications on the design of the high-level module, but also on the low-level one: the low-level one should be designed with the interaction in mind and it may be necessary to change its usage interface.



- Why DIP is important? well...
    - Dependency injection allows a client the flexibility of being configurable. Only the client's behavior is fixed. The client may act on anything that supports the intrinsic interface the client expects.
    - Dependency injection can be used to externalize a system's configuration details into configuration files, allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components. This includes, but is not limited to, testing.
    - Because dependency injection does not require any change in code behavior it can be applied to legacy code as a refactoring. The result is clients that are more independent and that are easier to unit test in isolation using stubs or mock objects that simulate other objects not under test. This ease of testing is often the first benefit noticed when using dependency injection.
    - Dependency injection allows a client to remove all knowledge of a concrete implementation that it needs to use. This helps isolate the client from the impact of design changes and defects. It promotes reusability, testability and maintainability.
    - Reduction of boilerplate code in the application objects, since all work to initialize or set up dependencies is handled by a provider component.
    - Dependency injection allows concurrent or independent development. Two developers can independently develop classes that use each other, while only needing to know the interface

    the classes will communicate through. Plugins are often developed by third party shops that never even talk to the developers who created the product that uses the plugins.

  - Dependency Injection decreases coupling between a class and its dependency.

- How did I implement it? I assured the DIP by keeping one level of inheritance, applying abstraction & reducing dependence between classes as much as possible.