**SOLID**

**Single Responsibility** – A class should have only one responsibility. It should only have one reason to change.

Open/Closed – Open for extension, Closed for Modification

**Liskov Substitution** – If class A is a subtype of class B. We should be able to replace B with A without disrupting the behaviour of program.

**Interface Segregation** – Larger interfaces should be split into smaller ones. Ensure that implementing classes only need to be concerned about the methods that are of interest to them.

**Dependency Inversion** – Decoupling and abstraction. High-level modules should not depend on low-level modules, both will depend on abstractions.

**ACID**

**Atomicity** – The entire transaction takes place at once or doesn’t happen at all.

**Consistency** – The database must be consistent before and after the transaction.

**Isolation** – Multiple transactions occur independently without interference

**Durability** – The changes of a successful transaction occurs even if the system failure occurs.

**CAP theorem**

Explain requirements in **distributed system with replication**

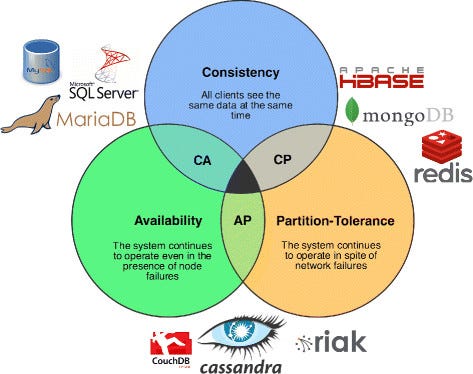
To make system designers aware of the trade-offs while designing network shared-data systems

CAP theorem – **It is not possible to guarantee all three of the desirable properties –** **consistency, availability, and partition tolerance at the same time in a distributed system with data replication.**

**Consistency**: Guarantees every node in a distributed cluster returns the same, most recent, and successful write.

**Availability**: each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. (every node (on either side of a network partition) must be able to respond in a reasonable amount of time.)

**Partition Tolerance**: system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other.



|  |  |
| --- | --- |
| **SQL** | **NoSQL** |
| RDBMS | Non-relational or distributed database system. |
| These databases have fixed or static or predefined schema | They have a dynamic schema |
| These databases are not suited for hierarchical data storage. | Suited for hierarchical data storage. |
| These databases are best suited for complex queries | Not good for complex queries |
| Vertically Scalable | Horizontally scalable |
| Follows ACID property | Follows CAP |
| Examples: MySQL, PostgreSQL, Oracle, MS-SQL Server, etc | Examples: MongoDB, HBase, Neo4j, Cassandra, etc |

**Reactive Programming**

Declarative programming paradigm that is based on the idea of asynchronous event processing and data streams.

Used in many areas such as, GUI programming, microservices, web programming, or reactive systems in general.

*Asynchronous processing means that the processing of an event does not block the processing of other events.*

**Design Patterns**

Solutions to common problems in software design

1. **Creational patterns**

Provide objection creation mechanisms that increase flexibility and reuse of existing code

Factory Method / Abstract Factory Method / Builder / Prototype / Singleton

1. **Structural patterns**

Explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient

Adapter / Bridge / Composite / Decorator / Façade / Flyweight / Proxy

1. **Behavioural patterns**

Take care of effective communication and assignment of responsibilities

Chain of responsibility / Command / Iterator / Mediator / Memento / Observer / State / Strategy / Template Method / Visitor

1. **Creational Patterns**

* **Factory Method**

Simplified version of abstract factory.

To decouple a client from a particular product that it uses. To relieve a client of responsibility for creating and configuring instances of a product.

// Interface for the product

interface Product {

void display();

}

// Concrete implementation of the product

class ConcreteProduct implements Product {

@Override

public void display() {

System.out.println("Inside ConcreteProduct::display() method.");

}

}

// Factory class responsible for creating objects

class Factory {

// Method to create objects based on input

public Product createProduct(String productType) {

if (productType.equalsIgnoreCase("Concrete")) {

return new ConcreteProduct();

}

// Add more cases for other product types if needed

return null;

}

}

* **Abstract Factory Method**

Consists of **AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, client**

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Lets you produce families of related objects without specifying their concrete classes.

Use the Abstract Factory pattern when clients must be decoupled from product classes. Especially useful for program configuration and modification.

* **Builder**
* **Prototype**
* **Singleton**

1. **Structural Patterns**
2. **Behavioural Patterns**

**Normalisation Forms in DBMS**

Minimizing redundancy from a relation or set of relations.

Redundancy may cause insertion, deletion, and update anomalies.

1NF

* A single cell must not hold more that one value (atomicity)
* There must be primary key for identification
* No duplicated rows or columns
* Each column must have only one value for each row in the table

2NF

* Already 1NF
* No partial dependency. That is all non-key attributes are fully dependent on a primary key

3NF

* Already be in 2NF
* Have no transitive partial dependency

Boyce-Codd Normal Form (BCNF)

* Already be in 3NF
* Each non key attribute only dependent only on the candidate key

4NF

* Table does not contain any multi-valued dependencies

5NF

* Decomposing a table into smaller tables to remove data redundancy and improve data integrity