Java technology stack

Java Core

OOP

**Encapsulation** – relationship

**Inheritance** – expand

**Polymorphism** – behavior

**Abstraction** – flexibility  
Benefits: Modularity / Reusability / Scalability / Maintainability

Other:

**Composition** - Objects can be composed of other objects. created from other objects

**Aggregation** - Objects can have references to other objects. weaker form of composition where objects are not dependent on each other.

**Modularity**: designed to be independent and reusable

**Loose coupling**: Classes should be loosely coupled to reduce dependencies

**High cohesion**: should have a single, well-defined purpose

**SOLID**

**Interface - Abstract class - Ordinary class – Enum**

Inner – logic group

Nested static – non init functions

Nested local – group without creation of diff class

Anonymous – create ghost code without name (if creation of local is expensive)

**Class** – abstract, final, static

**Meth** – abstract, final, static, native, synchronized

**Field** – static, final, transient, volatile

**Hash Code and Equals**

Reflexive – x == x

Symmetric – x == y || y == x

Transitive – x == y || y == z || z == x

Consistent – x == y || repeat x == t true

Non-null – null -> false

Annotations

+ **Code Clarity:** can make the code easier to read and maintain by reducing the boilerplate

+ **Compile-Time Error Checking**

+ **Flexibility:** allow make code configurations and reduce config files.

- **Runtime Overhead** (runtime reflection can lead to performance degradation)

- **Increased Complexity**

- **Dependency to framework**

**Min performance impact:**

Use Compile-Time Annotations / Cache Reflective Data / Limit Reflection /

Use Proxies and Code Generation

**Meta-annotation** is an annotation that is used to annotate other annotations.

@Target (Java elements to apply)

@Retention (source / class / runtime)

@Inherited (subclasses will inherit all annotations of parent)

@Documented – for java auto doc

@Repeatable – multiple annotations for the same element (ex. @Role or @Profile)

(c)Throwable -> (c)Exceptions | (un)Errors -> (un)Runtime | (c)IOexc

It's wrong to catch exceptions and do nothing with them

Try to avoid a critical situation in important points

Do not close the resource with your hands

Throw Early, Catch Later

Do not forget about log

First catch the bottleneck then the wide

IO – r/w at once. Thread saves

NIO – Buffer, Channel, Selector – Facter, more control, don’t block thread.

Serial \ Parallel (default) \ CMS \ G1

Immutable classes

classes whose objects cannot be modified after they are created.

+ Thread safety / simplifies code understanding and reusability / can improve performance

- Can be more difficult to create / Can be less flexible

**How to create**

- Make all fields final

- Make the constructor private

- Provide methods to create new objects

- Return defensive copies

String

Strings are **immutable**. Cannot be changed or modified

Pluses:   
+ Strings are immutable and do not need any **synchronization**  
+ String pool save memory and improve performance

Minuses:

- Each modification creates a new String object, and it can lead to performance issues

**String pool:** HashMap (key String Value + obj ref) contained all strings

Search in string pool is hash map search -> hashcode() + equals()

**String** for strings that won't change

**StringBuilder** for strings that will change in single-threaded system (faster that strings)

**StringBuffer** multi-threaded environment

Comparable vs Comparator

interfaces in Java that are used to compare objects  
**Comparable**:

- obj impl method `compareTo(T o)`

- **useful when you want to sort objects via Collections.sort()**

- natural ordering for the objects

**Comparator**:  
- obj impl method compare(T o1, T o2)

- mast be implemented as separate Comparator class

- **useful to sort objects using a custom comparison logic**

**-** more flexible and customizable sorting

Collections

Iterible > Collection -> List \\ Queue \\ Set

Map > Sorted map

ArrayList – classic (quick get, +- add, delete)

LinkedList – quick add, delete +- get

Queue – FIFO (linked)

PriorityQueue – FIFO

ArrayDeque – Head + tail

Set – store unique

HashSet – main implementation

LinkedHashSet – linked elements of set

TreeSet – red-black tree sort

HashMap - classic bucket key system

LinkedHashMap – classic + order

HashTable – thread save

TreeMap – like set, store key in red-black

Good HashMap Key

- Immutable (or risk map didn’t find key)

- Consistent (same fields) for equals() and hashCode()

- Effective hashCode() and equals() (do not return the same value)

Generics

**Main reason to** provide stronger type checks at compile time. Compiler uses this info to escape **ClassCastException**.

**Type erasure:** backward compatibility with older versions in which compiler removes type in compilation and used Object with type conversion instead

**Pluses:** Type Safety / Code Reusability / Elimination of Type Casting / Generic Algorithms Creation

**Minuses:**

- Because of **Type Erasure** information is not available at runtime and you cannot use instanceof, new, or create primitive array

- **Confusing Syntax**

- **Backward Compatibility** is also minus. In this case we can create collections without any contract

- **No Support for Primitive Types:** Wrapper classes only

**Type Parameter (T):** used when need specific type that will be determined later

**Wildcard (?):** used when object type doesn't matter or is unknown.

**PECS (Producer Extends, Consumer Super)   
In Generics:** principle in that helps to decide when to use a wildcard.

**As element of the system** defines the relationship between a system element and code

**Producer** - defines that the element produces something for the code. **And code read this data.**

**Consumer** defines that the element is consuming something from code. **And code gives some data**

**Bounded wildcards**

**<? extends T>** **upper bounded wildcard** - parameter will be T or any subtype of T.

<? extends Number> -> Number // Integer

Upper bounded lists **are immutable**

**Use** when you only need to get objects from a data structure. Because compiler can't determine which subtype is needed.

**<? super T>** **lower bounded wildcard** supertype of T or T.

<? super Integer> -> Integer // Number // Object

**Use** when you need to put objects in a data structure.

**Exceptions**:

Generic classes cannot extend Throwable // Cannot be thrown // Cannot be in catch

As Producers can handle exceptions for read state

**Reification** is the process by which an abstract idea about a feature becomes an explicit part of a program's structure.

But Java Generics are not reified

Serialization

**using the Serializable method:**

- specific marker interface

- writeObject on the io stream

- all subclasses need to be serializable

**using the Externalizable method:**

- marker interface with read/writeExternal methods

- create custom serialization and deserialization settings

Multithreading

allows multiple sequences of code (threads) to be run concurrently

**Thread scheduler** part of the JVM that decides which threads should run according to **thread priority** (1 – 10 d5)

**Thread state** NEW RUNNABLE BLOCKED (waiting for a monitor) WAITING (another thread wait)

TIMED\_WAITING TERMINATED

**Create Thread**

- extends the Thread class and override the run()  
- implements the Runnable interface

- implements the Callable interface

- Executor in java.util.concurrent and create **ThreadPool** (Thread Reuse // Control Over System Resources // Improved Stability)

- ThreadFactory in FixedThreadPool to customize the properties of the threads being created

Implementing Runnable//Callable is more flexible. Callable can return value and throw checked exception.

**Calling** the run() does not call a new thread, it simply calls the method.

- in a thread Thread(runnable).start();

- thread pool //executor.submit(runnable);

- current thread //runnable.run()

We can **stop thread** by interrupt() // while(flag) condition // Thread.stop() (deprecated and not recommended) // end of main thread (System exit)

**Deamon thread** JVM will not wait for daemon threads to complete before exits

**Problems**

- **Race conditions**: when one thread modifies variable other threads can’t see changes (use atomic operations and volatile vars)

- **Reordering and Happens before**: JMM can change instructions. (volatile or synchronized blocks)

**Strategies to achieve thread safety**

**- Immutable**: objects are thread-safe.

- **Synchronization** to access code from a single thread

- **Atomicity** for variable reads and writes

- **Thread local** variables

**Deadlock** - threads are blocked forever waiting for the other to release a lock. Very difficult to diagnose and fix

**Prevent Deadlock**

- Avoid Nested // Unnecessary Locks

- Use Lock Ordering // Lock Timeout

- **Debugging**: JVM or Thread dumps (get call stacks) or ThreadMXBean (get live threads)

**Exceptions** thrown in a thread cannot be caught in other threads.

- try-catch block inside the run()

- Thread.setUncaughtExceptionHandler

- Future in try catch

- use callable.call instead of run()

**Synchronized** (if you don't need specific conditions) created section where only one thread can execute at a time.

- **Methods block / static methods** - until the lock is released

- **Code Blocks** protect only part of a method

**wait(), notify(), and notifyAll()** allow

threads to communicate about the lock status of a resource.

**wait():** tells the current thread stop

**notify():** wakes up a single thread

**notifyAll():** wakes up all the threads

**Locks** is more flexible (If you need more advanced features)

- **Condition** Await() signal()

- **Locksport** basic synchronization methods **park unpark** that doesn’t throw exceptions

**ReentrantLock** allows one thread to get enter the monitor without stopping (lock, tryLock, unlock)

**ReadWriteLock** allows multiple threads to read shared data concurrently.

**StampedLock**

**Writing** (**exclusive lock**): only one thread can hold the write lock at a time.

**Reading** (**optimistic lock**): Multiple threads can hold the read lock if the write lock is not held. Write lock is blocked until all read locks are released.

(**pessimistic lock**): it is read, blocking other transactions to prevent conflicts

**Optimistic Reading:** If the data read is not modified while reading, the result is valid. If not, the operation is retried with a full read lock.

**java.util.concurrent API**

**Basic fiatures**

**Future** is an interface that represents the result of an asynchronous computation

isDone() // get() // get(timeout) // cancel()

**Atomic Variables:** atomic operations (incrementAndGet)

**ForkJoinPool** is a tool for parallelize tasks. Gives RecursiveTask and RecursiveAction objects with compute() method help to split job and how to combine the results  
- **fork():** This method asynchronously executes the task

- **join():** wait for a task to complete and obtain its result

**ScheduledExecutorService** can schedule tasks to run after a given delay

**ThreadGroup** structure that controls the state of multiple threads as a single unit

**Strategies**

**Semaphore** to limit the number of threadsthat can access resource

**CountDownLatch** initialized with number of completed threads (operations). Idea one or more threads wait other threads

**CyclicBarrier** allows fixed number of threads wait for each other.

**Phaser** for synchronizing threads that meet at a certain point of execution

// register() to add threads // arriveAndAwaitAdvance() wait for others // arriveAndDeregister() remove thread

**Exchanger** synchronization point. Waits until two separate threads call method

**CompletableFuture** in Java is a way to run tasks in the background, used for asynchronous programming.

**Concurrent Collections:**

Synchronized.collections (sList, sSet, sMap) (collection as monitor)

util. concurrent – ConcurrentHashMap (backet lock) CopyOnWriteArrayList (copy of collection with merge)

**Project Loom** OpenJDK project try to create "virtual-lightweight threads”

**Reflection**

allows you to inspect, modify, manipulate classes, methods, field at runtime.  
**Runtime Type Identification**: useful in serialization, cloning, etc.

**Dynamic invocation** of methods, obj and values and fields

**Extensibility Features:** Spring and Hibernate use reflection for things like dependency injection, creating proxies, and mapping objects to database records

Cons  
**Performance Overhead**:

**Security Restrictions**: (requires runtime permissions)  
Code complexity

**Breaking Encapsulation**(oop principles)

JVM JMM and GC

**Serial GC** - single-threaded collector. Stops all application threads (stop-the-world)   
Uses copying method (After 7 passes transfers objects from young to old)  
Requires less CPU time than other collectors. (Use when CPU use is more important than latency)

**Parallel GC** - multiple threads collector. Uses same copying method (Use in medium to large-sized datasets and pause times are not a concern)

**Concurrent Mark Sweep (CMS)** – concurrently marks dead object without stopping and delete them in short stop phase. (good choice for applications that require short pauses and can share CPU power with the garbage collector)

**G1 GC** - memory is divided into a set of uniformly sized heap regions. Uses defragmentation policy (Suitable for applications that require large heaps (>4GB), mixed GC pauses of predictable duration, and high throughput)

**JVM Tuning**  
- Heap Size (-Xmx and -Xms params)

- Stack Size (-Xss)

- Young Generation Size (-Xmn)

**GC Optimization**

- Selecting a Collector

- Adjusting GC Pauses

- Parallel GC Threads

- Trigger garbage collection

Classloader

**Need for** Dynamic modules loading / Hot deployment (without app stopping) / Security loading / Loading from nonstandard sources

**LC:**

1. Loading (Lazy by triggering and inmem init)

2. Linking

- Verification of the class file

- Preparation init mem for class

- Replace all objects with direct refs.

3. Initialization (when class is used for the first time) – access static fields / init blocks / ensures that superclass also init

4. Usage / 5. Deletion

**Types:**

1. Bootstrap: parent of all class loaders (It loads the core Java APIs) Part of JVM

2. Extension (loading extensions libs)

3. System/Application (It is responsible for loading all the app classes)

4. Custom

**Delegation model**

When a class loader is asked to load a class, it delegates the task to its parent class loader and load it only if parent can’t (preventing multi file versions)

**Custom class creation**

- Extend ClassLoader class

- Override findClass method

**If Class Already loaded**

it typically behaves like any standard classloader (return loaded from cash)

**How to reload**

- Create new Classloader instance (need to reload all classes)

- Create custom versioning to define changes and reload them

Java 8

**Default methods in interfaces**. This helps us add new features to existing implementations without too much trouble.

**Functional interfaces -** exactly one abstract method

**Lambda expressions** - don't have to spend time creating anonymous objects

**References to methods -** to simplify the code (Comparator, Runnable, Callable)

**Optional** - handy way avoid nullPE

**DateTimeApi** – new objects that help to work with time and date

**Stream vs Loop**

- feature of functional programming to write in declarative way. + Readability

- can be run in parallel (parallelStream())

- chaining code simplification

- can be quicker in parallel

**Minuses**

- can be challenging in debugging (peak)

- do not have side effects

- exception handling problems (wrap)

- can’t break to small part of code

**Choose loop** if you need single operations and have system side effects

Predicate interface

represents a predicate (a boolean-valued function). **Used** when you need to filter a collection or test a condition.   
**obj.stream().filter(PredicateObj)**

(FlatMap) - transforming multiple lists of lists to create a single list.

Java 9-17 features

LTS

- Java 17 (LTS) - September 2021

- Java 21 (JDK 21) — Released in September 2023

- next will be 25 (LTS)

**Java Records**

new kind of class declaration to hold immutable data

public record Point(int x, int y) { }

- immutable // readable // contains correct impl of equals / hashcode / toString

Java intern()

returns a canonical representation of the string object. JVM check if a String object with the same content exists in the pool, and return the existing instead of creating

Arrays.createList -> List.of/Set.of

Streams -> takeWhile/dropWhile

Private methods in interfaces

Var – method local variable

New Switch Case view like lambdas

case SUBSCRIBER -> true;

Null Pointer says what variable course the problem in logs

Multiline strings

Algorithms

Sorting algorithms

| time | space | datasets

Bubble Sort | O(n^2) | O(1) | small

Selection | O(n^2) | O(1) | small

Insertion | O(n^2) | O(1) | sorted or small

Merge Sort | O(nLOGn) | O(n) | large  
Quick Sort | O(nLOGn) | O(log n) | large  
Heap Sort | O(n log n) | O(1) | limited mem

Radix Sort | O(nk) | O(n + k) | known range

Counting | O(n + k) | O(k) | known range

Searching algorithms

Linear Search | O(n) | O(1) | unsorted

Binary Search | O(log n) | O(1) | sorted

Hash Table Lookup | O(1) | O(n) | by key Trie Search | O(m) | O(m) | tree-like

A\* Search | O(b^d) | O(b^d) | pathfinding

Testing

Functional \ Non-functional

Positive \ Negative

By access (black\white\grey)

**Manual** – low cost and high speed

**Auto** – high cost at start, safer in the end

**Unit** – one logical unit (desc. FIRST rule)

**Fast/Independent/Reliable/Self valid /Timeliness** (as soon as possible)

**Integration Testing –** logically chained unit

**System** – global application BL check

**Functional** – emulate user behavior

**Smoke testing** – test before app run

**Regression test** – run of already created test cases

**Acceptance test –** unit BL check

**Penetration test** – various stress tests

**Fuzzing test** – random input data

Test Pyramid is an abstraction that means grouping tests into different levels and show how many tests should be in each of these groups

TDD Coding as Production vs. Coding as Thinking (When there is no clear behavior but there is a set of input and output values)

Stub vs Mock vs Spy

**Stub** is a minimal interface implementing

returns hardcoded values

**Mock** fake implementation of an interface

**Spy** is real object, but its behavior can be controlled or inspected by the test

DB

SQL

is a programming language used to communicate and manipulate databases.

**DBMS** (Database Management System)

software that interacts with end users

**Relational** – defines database relationships as relations (Postgre, Ms)

**Non-relational** – relations between objects are represented as non-relational

Tables/Documented/Xml

SQL vs NoSQL

In cap SQL is **CA** **consistent** (all clients have the same view of data) and **available** (All clients can read and write)

NoSQL is **CP** (MongoDB) or **PA** (DynamoDB) Partition tolerant (Fail tolerant) and consistent or available (All clients can read and write)

- Data Structure (Table vs Various)

- Scalability (Vertical vs Horizontal)

- ACID (Full vs non-full)

CAP

Consistency - data in different replicas are the same

Availability – can get response to any request, but it can be not relevant

Partition tolerance - in case of unavailability of one of the sections, the system continues to work

CA – MySQL/Postgres

AP – Dynamo/Simple DB

CP – MondoDB/Redis

Execution plan is a kind of map that outlines how the SQL query will be executed by the database system

Common migration problems

**- Schema, Constraints and Data Type Incompatibilities**

**- Large Data Volume:** time-consuming

**- Data Consistency**

- **Code Conversion**: (JDBC in app)

**-** **Unexpected Performance Issues**

- **Vendor Lock-in** // **Training** // **Cost**

Normalizationis design rules to organize data to minimize redundancy and avoid data anomalies

- Eliminates duplicate data

- data consistency and flexibility

- improved performance and maintenance

**1NF –** Each table cell should contain a single value, and each record needs to be unique

**2NF** - all non-key attributes should be fully functionally dependent on the primary key

**3NF** - all non-key attributes should not depend on other non-key attributes

**4NF** - for any dependency A → B, A should be a super key. It means that A as key should uniquely identify each record in the table. (it can be pair of keys as a)

**5NF -** table should not have multi-valued dependencies. depends on another column (or set of columns), but not on the whole key.

ID COURCE TEXTBOOK

1 Math Math Book  
1 Math Calculator

This need to be separated to courses table and books table

**6NF** - the candidate keys should imply every join dependency. With candidate key with we can decompose and unite table without data loss

**Indexes**

is a data structure that improves the speed of data retrieval operations on a database table.

Database creates a data structure like hash map. Hashes key-value pair and store it in the buckets.

+ Fast sort and get operations

- Indexes take up storage space

- can slow down write operations (indexes need to be updated every time data is changed)

By type:

**Clustered Index** - determines the physical order data in a table. Only one per table. Created when a primary key is defined

**Non-Clustered Index:** logical data order. We can have multiple non-clustered indexes der table.

**Unique Index:** ensures that the index key contains only unique values. Can be clustered and non-clustered.

**Composite Index**: includes more than one column in the index key. useful when you run queries with multiple columns in the WHERE condition.

Indexes:

**B-Tree Indexes** - Most common and versatile type of index. Well-suited for data with a wide range of values. Worked like classic binary tree

**Hash Indexes** - hash function is used

to calculate the bucket location for the key. Best for: exact value of a key is known

**Bitmap Index** uses a bit (0 or 1) to represent whether a condition is true or false. Rapid way to find records in large table and not unique values

**Full-Text Index:** used to store and search for words or phrases within text data.

**Spatial (GiST) Index:** used for geo data types or geometric shapes

**GIN (Generalized Inverted Index)** is a type of index in PostgreSQL designed for full-text search and complex data types like arrays and JSON. It maps words or values to lists of document IDs where they appear. This allows for fast retrieval of data based on specific keywords.

Rules for Setting Indexes

1. Choose the right columns

Uniqueness // Selectivity (selected in most cases) // Frequency of use (queried in most cases) // Avoid indexing rarely used columns

2. data types

best on data types with fixed length and well-defined ordering (integers, dates, and strings)

3. Use B-tree indexes for most cases

4. Consider other index types for specific needs: Hash indexes for equality queries // gin for json // GiST for spatial data and geometric queries

5. Monitor index usage: like `pg\_stat\_all\_indexes` to find unused or inefficient indexes

6. Think about composite indexes only in case of specific very often used query

SQL Subsets (Data … Lang)

**DDL** (Definition) managing all table objects CREATE, ALTER, DROP

**DML** (Manipulation) managing all schema objects SELECT, INSERT

**DCL** (Control) controlling the permissions and access like GRANT and REVOKE

**TCL** (Transaction Control) COMMIT, ROLLBACK

View is a virtual table based on the result-set of an SQL statement (real-time data). But every time a view is queried, view run query to produce the result set.

Good for small queries and datasets

Materialized View is a physical copy, representation of the base table. Not need to run the underlying query but need to refresh the periodically to ensure that the data is up-to-date

ACIDa set of properties that guarantee that transactions will work correctly

**Atomicity –** transaction must work at once or be canceled fully

**Consistency –** data must comply with all table rules (validations, data type)

**Isolation –** parallel transactions should not provide influence on result of each transaction.

**Durability –** sure that when transaction is ended the result is saved

Isolation levels

**Read uncommitted** - Worst data consistency (highest speed) each transaction sees the uncommitted changes of another one (dirty read error)

**Read committed -** For this level, concurrently executing transactions see only committed changes from other transactions (repeatable read - reads the same row twice but gets different results)

**Repeatable read** - transactions can block other rows while reading (phantom read, transaction re-executes a query and gets a separate set of rows)

**Serializable** - Transactions are called strictly in chain. Highest data consistency, but the lowest performance speed.

Basic commands

**WHERE** – filter records before grouping

**(group by) HAVING** – filter records after grouping

**(From) GROUP BY** group the result-set by one or more columns

**(select) COUNT, MAX, MIN, SUM, AVG**

**(select) DISTINCT** return only unique values

**(from) ORDER BY … ASC|DESC** sort the result-set in ascending or descending order.

**(where) BETWEEN** – interval selection

**(where) LIKE** – use with **%** (none, one or more chars) and **\_** (one char) to find value by mask (ex. %day or \_\_\_-support)

**(where) IN** – find in multiple values

select score, **ROW\_NUMBER()** OVER (ORDER BY Score DESC) - This function assigns a unique row number to each row within result set. (Duplicated values work as common values)

**RANK():** rows with equal values, it assigns them the same rank, leaving a gap for the later ranks (1, 2, 2, 2, 5)

**DENSE\_RANK():** does not skip ranks for the rows with equal values (1, 2, 2, 2, 3)

SQL JOIN operation

**(From) INNER JOIN \_table\_ ON** returnrows that have matching values in both tables

**OUTER JOIN** – return inclusive data

**LEFT JOIN** - all rows from the left table and the matched rows from the right table

**RIGHT JOIN** - returns all rows from the right table, and matching rows from the left table

**FULL JOIN** - combines the results of both left and right joins

Subqueries

help in dividing complex queries into simpler, more manageable parts

- improves readability and facilitates debugging

**Single-row Subquery:** (in where condition)

**Multiple-column Subquery**: (used in the from clause)

**Correlated Subquery**: (Where uses values from the outer query)

Stored procedures

set of SQL statements with an assigned name. (CREATE PROCEDURE // EXEC)

**Use** them when you need:

complex SQL // enhance security // reduce network traffic // improve performance and reusability

**Cons:** can be database-specific //Handling of errors and debugging can be difficult //

Trigger

procedural code automatically executed by event.

Types: **DML Triggers** (triggers on data modification like INSERT, UPDATE)

**DDL Triggers** (on table modification CREATE, ALTER, and DROP)

Event types: **BEFORE/AFTER** event and **INSTEAD OF** event (replace the triggering event)

**Pros:**

- can enforce complex business rules

- encapsulate logic in the database

- maintain data integrity and validity

**Cons:**

- can lead to unexpected results

- can negatively impact performance

- can be hard to debug and troubleshoot

Cursor

used to retrieve rows from a result set one at a time. (pointer in result set row)

LC: DECLARE // OPEN // FETCH NEXT // @@FETCH\_STATUS (0 if last, -1 if null) // CLOSE // DEALLOCATE (delete memory)

Pros: complex logic and previous row operations logic

Cons: slower and more mem usage

Handle DB schema evolution

Managed using Flyway or Liquibase.

PostgreSQL

TOAST mechanism

allows it to manage large data values more than 8KB.

**Mechanism of work:**

- Large Data Arrives (around 8KB)

- TOAST checks If data can be compressed to 8KB, if not the data is divided into chunks. (about 2KB by default)

- Chunks Are Stored In TOAST Table: Each chunk has a pointer that keeps track order of the chunks.

- Original Table Stores TOAST Value (TOAST ID and TOAST table)

**But Too much TOAST data can lead:**

- longer backup times

- increased disk usage

- longer query times

- difficulties in managing the instance.

NoSQL

General

database architecture designed to store, retrieve, and manage data in nonclassical ways

- **Document-based** (such as MongoDB), **Key-value pairs** (such as Redis), **Wide-column stores** (store data in cells that are grouped in columns) (such as Cassandra), **Graph-based** (such as Neo4j).

- have dynamic schemas for unstructured data

- horizontal scaling

- prioritize performance and scalability

over transactional consistency

**Good choice**: dealing with large volumes of unstructured data, require high write loads, or need to scale horizontally

**Pros**: Highly scalable and distributed

Flexible data model // High performance

**Cons**: Limited transactional consistency

Low query capabilities

How to choose NoSQL type

**Core parameters**: data structure size, nature of your queries, size of your data

**Document Store**: great when the data can be grouped together (ex. JSON)

**Key-Value**: for caching solutions, session management, and maintaining user profiles

**Column Store**: for Big Data solutions where a written load is heavy, and the data can be grouped into loose clusters

**Graph-Based**: scenario when relationships are important. (social networks, recommendation engines)

NoSQL data consistency

**Eventual Consistency** - allow temporary inconsistencies, but guarantee that the system will eventually become consistent

**Tunable Consistency**: allows you to choose the level of consistency vs performance for each transaction.

**Using Transactions:** not all NoSQL databases support

**Sharding Strategy**: distributed across multiple servers

**Read & Write Concern**: can set specific 'read' and 'write' concern levels to ensure consistency (dictates how many replicas must confirm operation)

**Using Change Data Capture**: (like Apache Kafka) corresponding schema changes in real-time events

Indexing

**Document Store** – index any field in a document

**Key-Value Store** — the key generally indexes database

**Column Store** - primary key is mandatory index key for each table

**Graph Databases** – relation as key

Sharding and Replication

**Sharding** - splitting and storing a single logical dataset in multiple databases.

(can greatly improve the performance of a database because operations are executed on a smaller subset of data)

**Replication** - maintaining copies of the same data on multiple machines. (improve data availability and durability)

NoSQL Security

Authentication / Role-based access control (RBAC) / Encryption / Auditing / Backups

NoSQL Performance Optimization

Sharding / Indexing / Denormalization / Caching / Tuning Configurations / Compression / Query Optimization / Monitoring

CAP

**Consistency** - data in different replicas are the same

**Availability** – can get response to any request, but it can be not relevant

**Partition tolerance** - in case of unavailability of one of the sections, the system continues to work

CA – MySQL/Postgres

AP – Dynamo/Simple DB

CP – MondoDB/Redis

REST

GET/HEAD/POST/PUT/DELETE/OPTIONS (list of possible server commands)

Rest rules

**Client-Server** - client and server

should be separated from each other and be able to develop individually.

**Stateless -** The server should not remember the state. information for the transaction work is transmitted in each request.

**Caching –** we can cash repeatable request to save performance

**Unified Interface** – all requests must be build using a common template. + HATEOAS + selfDescriptive

HATEOAS rules:

- **Server sends URLs for all actions**

- **No hard code information**: provide all URLs dynamically

- **Links indicate state** (if we can’t delete resource no need to show delete link)

- **Use of MIME types**

- **application/json** (default)

- **application/hal+json** (HAL (Hypertext Application Language) easy way to hyperlink between resources in your API)

- **application/vnd.siren+json** (specification for representing entities, including relations with other entities and actions)

**Multilevel system** - means that the client does not know if the server which responds, the end server

**Ease of presentation** – response format should not change

An HTTP method is idempotent if a repeated request, done one or more times in a row does not change the state of the server. (POST not)

Safe methods are those that do not modify internal resources. (OPTIONS, GET, HEAD)

The Richardson Maturity Model

useful for designing, developing, communicating about, and making decisions regarding your RESTful APIs.

**Level 0** — HTTP only as a transport system (SOAP-based web services)

**Level 1** — Resources: uses only HTTP POST for all operations

**Level 2** — HTTP Verbs: This level uses all HTTP methods and return errors

**Level 3** — Use of Hypermedia Controls (HATEOAS): return the information about how to interact with the API

Rest best practices

must accept and respond in one format

return errors status codes with msg in error body

don’t use verbs

use plural nouns

don’t forget about documentation

use recourse nesting /users/+id/+orders

SOAP is a protocol that is used to implement web services

REST is a set of architectural rules

REST can use SOAP, SOAP can’t use REST

- SOAP xml only

- and can't be cached

- runs slower

- not as easy to test as REST

+ SOAP is used when it is not possible to use rest (requirement of stateful)

+ Support for legacy systems

+ great level of security (like the example of PayPal on soap)

WEB socket – suitable for direct access, such as games, they have only body

It can be not a client server. Server can ask client a question. Saves session state

- depend on IP and port

- only vertical scailing

- requires memory for storing data

Rest security

**Transit Level**

- Use Secure Protocols (HTTPS for web, SFTP for file, TLS for email)

- SSL/TLS Certificates to provide encrypted connection

- VPN

**Rest level**

- Encryption // - Access Controls // - Regular Audits // - Secure Backups

**Both**

API Key Management that used to authenticate a user and control API usage (do not provide protection against attack) (generation, storage, rotation)

// Data Minimization (minimum amount of sensitive data)

SOAP

(Simple Object Access Protocol) is a protocol used for exchanging data in web services using XML.

In Java, SOAP web services are implemented using the Java API for XML Web Services (JAX-WS).

1. **Add dependencies**

2. **Define the Service Endpoint Interface** (with the help of @WebService and @WebMethod annotations)

3. **Implement the Service Endpoint Interface** (service wrote in step 2)

4. **Publish the SOAP web service** (javax.xml.ws.Endpoint.publish())

Hibernate

Framework to interact with databases

- **Simplified CRUD Operations**

- **Easy to switch databases**

- **Improved development speed**

- **Cache Management**

- **Mapping Flexibility** (supports various relationships like 1-2-1 1-2-many)

- **Transaction Management**

Minuses:

- **Performance Overhead**

- **High threshold for entry**

- **Lack of Control:** less control compared to traditional JDBC.

- **Unsuitable for Simple or Complex Queries**. For simple queries might be an overkill. Also not be suitable for complex queries or stored procedures

JDBC vs JPA vs Hibernate

**Spring JDBC:** simplified approach for simple models

**Hibernate:** offers ORM features and abstractions for complex models

**Spring Data JPA**: For complex models but better fits with Spring ecosystem

Caching

**First-level cache (session)**: local hibernate cash, private to each session. (use this specific to a particular session)

**Second-level cache (query cache)**: stores the results of queries, can be shared across multiple sessions  
(use per query)  
**Collection cache**: collections of entities, such as lists and sets (use per entity list)

**Natural ID cache**: Caches entities based on their natural identifier (entity with unique identifier)

Build Tools

Maven

Minuses:   
- Complexity // - Inflexibility //   
- Slower than analogs because of full lifecycle   
 - hard to debug deployment issues   
- Lack of rollback functionality

Spring

Spring

+ large set of infrastructures and libs

+ simplifies Java EE complexities

+ Open Source // Testability

+ Inversion of Control (IoC)

+ Transaction Management

+ Integration with Other Frameworks

- Complexity // - High barrier to entry

- Numerous Configurations

- Performance: sometimes cause minor performance degradation

- Multiple Ways to Accomplish a Task

- this is framework

IOC/DI/DL

IoC is an architectural solution - when the programmer delegates the work of managing objects to framework

Consists of (BeanFactory, Application Context)

DI - @Autowired when spring injects itself

DL - getBean() when we can manually pick up this dependency for use

Bean Type

**Singleton** – like pattern (control by Sprig)

**Prototype** – single object (creation control

**Request** – bean scoped to HTTP req

**Session Beans**

Session – http session

G-session – global http session

**Application Beans**: ServletContext LC

**WebSocket Beans**

**Custom bean scope**: implement Scope and control init and destroy flow

Main annotations

**@DependsOn** - define a dependency between Spring beans to ensure that a certain bean is initialized before another

**@Conditional** - component or configuration should only be registered if the specified conditions match   
(Condition class need to implement Condition interface)

- Environment-specific beans

- Optional components

- Feature flags

- Hardware-specific beans

**@Autowired** (fields, method, or m init params, constructor) @Quilifier @Lazy @Bean , required=false

**@Primary** if Autowiring conflict, annotated bean gets the priority

If you use both @Primary and Qualifier, @Qualifier takes lead

**@Resource** – is a part of JSR. Performs injection by name and looks for a bean with the same name in field or setter method. Will not throw any exception in case if bean is not found. (will throw if you specify lookup="beanName")

Bean LC

- Obj Constructor

- Dependency injection \*Autowiring

- Calling Aware interfaces to define beans metadata (example BeanNameAware, BeanFactoryAware)

- initialization step

@PostCostruct (part of Java JSR)

- Initializing default values

- Starting background tasks

- Loading data

initMethod

afterPropertiesSet ()

- post init (BeanPostProssesor.postProcAfterInit())

**Destroy LC**

- @PreDestroy (part of Java JSR)

- Releasing resources

- Stopping background tasks

- Saving data

- destroyMethod

- DisposableBean.destroy()

Spring Configuration Types

**XML based configuration**

+ the oldest method that has many examples and some tricky attributes

+ concentrated in 1 or more xml files

= this is xml, high entry threshold you need to know properties

= you will see errors only when you start the application

**Annotation based configuration:**

instead of using xml inside xml it is described that the context will be annotation-config. Then use annotations (@Component, @Service, @Repo

ComponentScan, and so on)

+ Simplicity of use \ decentralized

**Java based configuration**

Annotated the class as @Configuration, then annotate the object creation methods using the @Bean annotation

+ centralization \ + Custom logic

Event listeners

+ Loose Coupling // Ease of Use

+ Synchronous and Async Support

+ Extensibility (Easy to extend)

- Hidden Behavior

- Synchronous Performance: slow down if you have long-running tasks

- Debugging and Testing: more difficult

1. class extending ApplicationEvent

2. Use ApplicationEventPublisher to publish events (part of any Service)

3. Create an Event Listener with @EventListener

**AOP**

provides the ability to dynamically add logic around the actual logic with simple functions.

Advice – when new logic is called

Aspect - class analog (store advices)

Joinpoint - (like annotation) – point where advice will call

Pointcut - (condition) – choose of the necessary advice

Spring AOP and AspectJ logically separated

Spring AOP gives us common AOP solution for beans

AspectJ gives us complete AOP solution

Spring boot is a Spring module that provides a RAD feature

(Rapid Application Development)

Maven deps/@SpringBootApplication + @ComponentScan to class/@Component or @RestController

@Transactional - (2 phase commit) any DataAcsessObject exception will automatically roll back previous changes

Spring JDBC

JDBC is a part of the Java EE

**Provides JDBC abstraction**

**Connection Management:** automates database connection, preparing statements, and handling transactions

**Exception Handling:** translates standard SQL to Spring's exceptions

**Query Execution and Result Set:** provides methods to execute SQL queries, updates, and stored procedures.

**Resource Management:** JdbcTemplate automatically closes the resources

JDBC vs JPA vs Hibernate

**Spring JDBC:** simplified approach for simple models

**Hibernate:** offers ORM features and abstractions for complex models

**Spring Data JPA**: For complex models but better fits with Spring ecosystem

Problems if they are using together

**Transaction Management** - if both technologies are used within the same transaction. It can lead to problems

**Code Complexity // Performance**

**Consistency in Codebase**

**JPA and JDBC handle exceptions differently**

Transactions

**PlatformTransactionManager** – by code (commit() rollback() methods)

**@Transactional** - Spring ensures method will be run within a transaction

Works with the help of Spring AOP

Preferable for clean core

@Transactional attributes  
**Propagation**: how transactions relate to each other

- REQUIRED (reuse existing transaction or start new one)

- REQUIRES\_NEW (new transaction)

- SUPPORTS (reuse or execute non-transactional)

**Isolation // Timeout // Read-only flag**

**Rollback Rules:** by default, roll back on runtime exceptions or customized with rollbackFor and noRollbackFor

SQL exceptions handling

translates standard SQL to Spring's exceptions by the help of

- SQLStateSQLExceptionTranslator

- SQLErrorCodeSQLExceptionTranslator

Map ResultSet to Object

**By code** - RowMapper interface with mapRow() method

**By Spring** – BeanPropertyRowMapper when column names match target object field names

JdbcTemplate vs NamedParameterJdbcTempl

**JdbcTemplate**: Parameters provided as placeholders using '?' in queryForObject method

Works fine with small amount of params

**NamedParameterJdbcTemplate**: uses SqlParameterSource interface (map or bean implementations) or simple Map to provide named params params.addValue("id", id);

improves code readability and maintainability for big amount of params

Configure DataSource in Spring JDBC

1. **Java-Based**: declare a `@Bean` method that returns a `DataSource` example HikariCP as the connection pool

2. **XML-Based:** using DBCP (Database Connection Pool)

How Spring JDBC handle connection pooling

via a DataSource object or connection pull implementations like HikariCP, Apache DBCP, or C3P0

Handle SQL NULL values

Default values for primitives

Null for Object types

Large datasets

**- PagingAndSortingRepository** interface: fetch data in smaller chunks

- **Control Fetch Size of a result set**

- **Use Appropriate SQL** (skip unnecessary data)

- **Batch Processing** (group of SQL queries as one)

If we don't want to store all data in memory:

**ResultSetExtractor + Streaming** (but other operations with this connection will be closed)

LOB data

The LobHandler interface in Spring JDBC is used to handle LOB fields

Stored procedures and functions

**SimpleJdbcCall**: simple approach

**extend StoredProcedure**: more execution control

DB connection retries policy

DataSource Connection Pool

Manual Retries

Spring Retry (doWithRetry method)

Spring JPA

JPA is a part of the Java EE

**Spring JPA is the Repository abstraction** and work with a database in an interface-driven manner

@Entity – POJO classes

**@Id** annotation

**Column Annotations:** If a field's name doesn't match

**Relationship annotations**: @OneToMany, @ManyToOne

EntityManager – provides persistence operations on entities

Repositories - provides implementation of data access layers

public interface EmployeeRepository extends JpaRepository<Employee, Long> {}

@Transactional - define transaction boundaries, propagation, isolation. Attributes same as in JDBC

**Propagation**: how transactions relate to each other

- REQUIRED (reuse existing transaction or start new one)

- REQUIRES\_NEW (new transaction)

- SUPPORTS (reuse or execute non-transactional)

**Isolation // Timeout // Read-only flag**

**Rollback Rules:** by default, roll back on runtime exceptions or customized with rollbackFor and noRollbackFor

**TransactionManager**: bean to be used for transaction management

Exceptions handling

translates JPA checked exceptions into unchecked exceptions by PersistenceExceptionTranslator

List of basic operations

Save // delete // count // find // delete //

…ById(ID id) // …All()

@Query annotation

To perform custom SQL queries in JPA. Params can be set as **“?1”** (means method's first parameter) or use **@Param("name")** on method input

Also, query has **nativeQuery** attribute used for SQL query instead JPQL

JPQL query is used FROM param as Java object

SQL used FROM param as database table

**@NamedQuery and @NamedQueries** are used to define static queries directly on the entity classes

Eager vs Lazy Loading

**Eager Loading** - related entities will be loaded immediately

\*ToOne relations

frequently access related data and/or work with small datasets

**Lazy Loading** – related entities will be loaded when the application accesses them for the first time

\*ToMany relations

large datasets and/or access-related data occasionally

Cascading operations

With a distributed from the parent to child (or children in oneToMany)

Example. @ManyToOne(cascade = CascadeType.ALL)

**All** // **PERSIST** save parent and all children (Insert)

**MERGE:** If the state of parent entity is updated, child will also get updated

**REFRESH:** If we reset the state of an entity, child will also get refreshed.

**REMOVE:** Delete operation

**DETACH:** Remove the association between a parent and child.

@JoinColumn / @JoinTable / @SecondaryTable

**@JoinColumn** specify the column for joining in single one-to-many direction

**@JoinTable** specify a join table in many-to-many relationships

**@SecondaryTable** used to map an entity to a secondary table

@Version

automatically incrementing a version field each time an entity is updated and causing transactions that have an outdated version number to fail.

Composite primary keys

**@IdClass** (extra class is needed)

**@Embeddable class** is created to hold the composite key and this is included in the entity class using the **@EmbeddedId**

Pageable and Sort

**Pageable** is an interface that contains requested page information return Page object as resultSet

**Sort** object that holds sort order and the properties to sort by

JPA Auditing

1. @EnableJpaAuditing on class

2. @CreatedDate or @CreatedBy or etc. to fields

Or

2. extend Auditable or AbstractAuditable

3. Implement AuditorAware in bean

@EntityListeners

used to hook into entity lifecycle events

like pre-persist (before save), post-persist

JDBC vs JPA vs Hibernate

**Spring JDBC:** simplified approach for simple models

**Hibernate:** offers ORM features and abstractions for complex models

**Spring Data JPA**: For complex models but better fits with Spring ecosystem

Spring MVC

is a module of the Spring Framework implementing the Model-View-Controller design pattern

**Model:** app data and business logic.

**View**: rendering the model data (HTML, CSS)

**Controller:** user interactions (HTTP requests, business logic execution)

DispatcherServlet is a servlet that receives all HTTP requests and dispatching them to the controllers

- **Request Handling**

- **Controller Selection** (via HandlerMapping defined by developer or automatically via component scanning)

- **View Resolution** After the controller has processed takes the logical view name and map to view via ViewResolver

- **Request Transformation** transform model and view into http response

- **Exception Handling** using HandlerExceptionResolver map exception to error view

Main annotations

**@RequestMapping** (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping)

**@PathVariable**

**@RequestBody @ResponseBody**

@ExceptionHandler

Spring Security

framework that provides security services for

**Authentication // Authorization // Protection against Attacks // Servlet API Integration // Security Propagation // Session management // Exception handling // OAuth 2 // JWT // 2FA**

Authentication

process of validating the identity of a user

**AuthenticationManager** interface encapsulate the authentication logic

**AuthenticationProvider** instances for authentication support for different types

Authorization

Process of determining user privileges

- Spring is using **method-level** security

**- URL-based** security (in configuration)

**@EnableGlobalMethodSec**

**@Secured("ROLE\_ADMIN")** to lock method by role

**@PreAuthorize("hasRole('ROLE\_ADMIN') or #user.name == …)** same but with Spring Expression Language

@EnableGlobalAuth

used to enable the Spring Security's global method security mechanism (but it is deprecated in 5.0 version

**@EnableWebSecurity**

annotation marks a configuration class as a Spring Security configuration

@AuthenticationPrincipal

inject the authenticated user into parameter without manually having to retrieve them from the SecurityContext.

Password encryption

**PasswordEncoder** interface with implementations for encoding schemes

SecurityContextHolder

is where the system stores details about the authenticated user.

Uses **SecurityContextHolderStrategy** to specify strategy. **ThreadLocal** by default

Session management

**Session Creation Policy**: control when sessions are created

**Concurrent Session Control:** limit of number of user sessions

**Session Fixation Arrack Protection:**

**Session Timeout:**

**Remember-Me:** remember-me authentication when user remain session

Session management works only for stateful apps

Exception Handling

Spring provides custom exceptions and handlers

**- http.exceptionHandling()** this lib contains **AuthenticationEntryPoint** and **accessDeniedHandler**

**- @ControllerAdvice and @ExceptionHandler** AOP across the whole application

Implementing OAuth2

1. Add Dependencies

2. Register Your App with provider (Google, Git, etc.) provider will then generate a Client ID and Secret

3. Add generated creds to property file

4. Implement controller and Security Configuration

Add @EnableResourceServer

Implementing JSON Web Tokens (JWT)

stateless authentication mechanisms for REST services

1. Add dependencies like JJWT

2. Implement a Filter (get token from request header + validate token in SecurityContext)

3. Add filter to Security Configuration

Two-Factor Authentication (2FA)

1. Classic Authentication: if pass valid create a "pre-authenticated" session

2. Two-Factor Code Generation (generated code usually stored in the database)

3. Two-Factor Code Input

4. Fully Authenticate the User

Password reset functionality

1. application verifies if the user email exists in the database

2. Generate Reset Token

3. Send Reset Link via email

4. User Clicks the Link and Resets Password

MQ

RabbitMQ

open-source message-queueing software dispatching messages between producers and consumers

**Steps**: - producer sends messages to an exchange

- messages routes to queues based on rules

- Consumers subscribe to queues and receive messages

**Why**: creation of asynchronously work system

**Pros**: Provides communication between microservices.

- Supports a wide range of protocols

- Can route to multiple consumers.

**Cons: -** Could be overkill

- management and Configuring for high distributive applications can be complex

Kafka

Apache Kafka is a distributed streaming platform for data pipelines and streaming apps. (like kinesis and elastic cash redis)

+ High Throughput // Fault-Tolerant (replicas) // Low Latency (by design) // Durability (distributed commit log) // Scalability

Web Servers

Tomcat

Tomcat, is an open-source web server and servlet container

- Serve Java Web Applications: As a servlet container,

- Web Server: serve static web pages

- Embedded Server: Tomcat can be embedded into a Java application

- Run WebSocket-based Applications

Jetty

is an open-source, lightweight, and highly scalable Java-based web server, and servlet container.

- Web Oriented Architecture Web Server: support for WebSocket, Asynchronous HTTP, and various protocols

- Embedded Server:

- Scalable Web Server:

- Development and Testing

GIT

Version Control System

tracks changes to a file or set of files over time and allows you to revert / see changes

- Speed and efficiency

- Distributed nature (can be offline)

- Branching and merging

- Large community and support

GIT

**.git** **repository** folder contains all the info that Git needs to track Objects // Refs // HEAD // config // desc // hooks // indexes

**Git config** - settings for Git

Global config for all repositories // Local config for specific repo

To see all setting you can   
use git config --global --list

**Sections**: Working directory (local) // Staging area (where you store files before commit) // Repository (where history of your project is stored)

**Git File States:** Untracked // Unmodified // Modified // Staged

**How Git tack file changes?**

**Content-addressable storage system**. stores a unique identifier generated by **hashing algo** for each version of a file.

Fetch vs Pull

**Fetch** - downloads the latest changes from the remote repository. Does not automatically merge them into your current branch.

**Pull** - combination of `fetch` and `merge`

Merge and Rebase

**Merge** creates a new commit that combines the changes from two branches. Creates a "merge commit" commit. (Provides a clear audit trail of changes.)

**Rebase** replays the commits from one branch on top of another branch. This rewrites the history of the branch being rebased, making it appear as if the commits were made on the target branch from the beginning. (clean up the history or agreed on this workflow)

Forks vs Branches

Forks owned by a different user or organization. useful for experimenting with new features. Forks are independent of the original repository

Dev Ops tech stack

IP netmask (/24)

is used to identify the network portion of an IP address. Helps to divide a network into subnetworks or subnets

Example, /24 subnet mask uses the first 24 bits for the network address and leaves the last 8 bits for host addresses.

Shell scripting

Basic commands

- **#! /bin/bash** - script interpreter (always on top)

- **VARNAME="Hello, World!"**

- **$(command) or `command`** - run the command and replace itself with the command’s output

- **if, elif, else, and fi** – conditional

- **for/ I in {} do … done // while** – loops

- **case** – function // funcname()

- **read** - Reads user input.

- **echo // exit**

CI/CD

Version Control   
Way of committing changes to shared repository.

Continuous Integration  
Automated tools that build newly committed code help find and address bugs quicker, improve software quality.

Continuous Delivery  
This stage involves testing the code in a production-like environment, which may include UI testing, load testing, integration testing, etc.   
The goal is to ensure that we have a deployment-ready code

CI/CD tools

Git -

Maven/Gradle – for Build/Testing/Document creation

Jenkins/Bamboo/TeamCity – automate build

JUnit/Mockito – for testing

Docker/Kubernetes – for containerization and orchestration

Jenkins/Bamboo/GoCD - automate the deployment process

Prometheus/Grafana – for performance statistics

Ensure zero-downtime deployments

Blue/Green Deployment

two identical production environments

Canary Releases

send 10% of traffic to blue envs, test,   
and send others after some time limit

How to store properties

Environment/System Variables: that a stored in the env by itself  
Configuration Files: files can be loaded at runtime based on the current environment  
Secrets Management Tools: Vault, AWS Secrets Manager, or Azure Key Vault

Troubleshoot of CI/CD

- Local Reproduction in case if it is not a local problem  
- Check the Error Message or Error log  
- various checks based on error message  
 - Check the pom.xml File  
 - Check the Dependency Repositories  
 - Check the Resources  
 - Check the Network

Key metrics

- Deployment Frequency  
- Failure Rate  
- Mean Time to Recovery  
- Test Automation Rate  
- Code Coverage

How to minimize build time

- Parallel Builds (main idea to split test and build processes)  
- CI/CD Branching (test new feature in lighter dev branch)  
- Performance, Accessibility Testing  
- Automated Security Testing  
- Optimize Dependencies (use only needed)  
- Incremental Builds (like in AWS CloudFront template)  
- Distributed Builds across multiple machines

Jenkins CI

open-source tool for CI/CD (like AWS pipeline)

- checking repositories for changes

- build code // - triggering tests // - deployment // - error notification

Pros: **open-source** (community // docs // frameworks // cases)

**Can be used across platforms** (lots of plugins to work with AWS, docker, etc.)

Cons: **Initial setup can be complex**

**- The UI is not very intuitive**.

**- Scaling Jenkins for larger codebases can be challenging**

AWS

Basics

**EC2:** On Demand / Reserved / Spot / Dedicated.

**AMI** (Amazon Machine Images)

**DB**

**AuroraDB / DinamoDB**

**Athena:** interactive query service

**CI/CD**

code pipeline/ code deploy/ code build / code commit.

Infrastructure as code

**CloudFormation:** YAML or JSON templates for set up your AWS resources.

**Elastic Beanstalk:** automatically manages quickly deploy of applications

**Lambdas:** event driving processors

Each lambda represents an image and a container. And Amazon takes over the work of all containers and their management.

**Amplify**: group of services designed to create full-stack apps

Cashing

**ElasticCache:** allows you to add a cache for frequently read data.

- Memcached: for simple cashing

- Redis: for more functions

Traffic switch

**Elastic Load Balancer:** helps balance traffic between several downstream instances

**Instance Autoscaling**

All at once: all stopped – all redeployed

Rolling: separate all instances into batches. Deploy half, then another

**Rolling with additional batch**

Canary switch: switch 10% of traffic to new instance than after limited period switch other traffic

Linear switch: switch % of traffic every n minutes.

**Security:**

**Identity and Access Management (IAM):** provides securely control

**Web Access Firewall:** protect against common network exploits

**Macie:** AI data security service

Systems Manager Parameter Store: for system parameters

Secrets Manager with KMS: for secure generation and containing secrets

Networking

Virtual Private Cloud (VPC): isolated private cloud

From private subnet:

Lambda ---> [(private subnet) ---> (ENI (Elastic Network Interface)) ---> (Destination (example S3)) ]  
To private subnet:

1) -> NAT (Network Address Translation) with IGW (InternetGetWay) -> www (or) aws s3

2) -> private VPC endpoint -> aws s3

Monitoring:

**X-Ray:** Debugging analyzing service

**CloudWatch:** monitoring service

**CloudTrail:** monitoring user activity

CloudWatch Evidently: feature for performance monitoring

Web content

**CloudFront:** speeds up distribution

of your static and dynamic web content

Data processing

**Kinesis**: collect process and

analyze streaming data in real time

Simple Notification Service: notify all subscribers

Simple Queue Service:containing messages

Containers

Fargate: service to run containerized applications

**Elastic Container Service:** will run your counterfeiters on clusters of virtual EC2

- can use Fargate for serverless containers

- highly scalable // - fault tolerant // - easy to update

Docker

The main idea behind Docker is to create

independent and isolated lightweight and fast environments.

Docker image: immutable file that contains application snapshot and dependencies

Docker container: is a running instance of an image

Docker volumes are directory on the host machine that is controlled by container.

Volume Drivers allow volumes to be hosted on remote hosts

+ Isolation — containers are isolated from each other  
+ Consistency — we have the same env during development  
+ Scalability — can be easily scaled up or down  
+ Portability — run on any system

+ Efficiency: Docker is lightweight and fast.

+ Helps with CI process  
  
- Complexity of knowledge and integration: can be a problem for who are new to Docker

- Security – If docker containers are runed from root and compromised,

there’s a risk that this could gain

access to the host system.  
- Framework Dependency. Docker still under development and may not have some features

- Every container need backup and recovery strategy in case of shout down

- Performance: Docker by itself will use as much system resources as the system will allow.

Alternatives

Podman // Containerd // Rancher

Backup and recovery strategies

- Committing Containers to an Image:

// docker container commit //

to store container filesystem changes

and container’s configuration

- Backing Up Volumes Separately

- Pushing Images or save as tar file

- Re-creating Containers from Docker hub and Local tar

Docker vs EC2 vs Virtual Env:

- Docker is a part of machine that uses reserved resources (in that concept on one ec2 can be stored lots of containers)

- Containers are lightweight piece of software that contains all parts to run application

- Docker containers are less resource-intensive than virtual machines.

- Docker containers are portable

Dockerfile is a text document that contains all the commands to create an image

Docker Layers- set of changes that have been made to the file system. layers are stacked on top of each other to create the final Docker image. If you make changes Docker will rebuild only the changed layer and all layers after it.

If changed

Dockerfile - layers defined after that line will be built again.

Image – container will use new image, running one will use old

Source Code - Docker itself does not automatically react to changes   
in your source code. But Docker volumes

can.

Container

Create // Start // Running //

Stop - sends a SIGTERM command, if not stopped after some period sends SIGKILL to terminate

Restart (stop + start) (useful for applying new changes)

Pause / Unpause - freezing all its processes (useful for troubleshooting or resource management)

Remove – remove stopped container

Docker Compose is a tool for defining and running multi-container Docker applications

List of commands

- docker run create and start a container from an image.  
- docker ps: list the running containers.  
- docker stop  
- docker rm  
- docker images - lists the images.  
- docker rmi - removes one or more images.  
- docker pull - pulls an image from repository to local machine (git fetch)  
- docker build  
- docker login - logs in to a Docker registry.  
- docker network create  
- docker volume create

Docker registry // Docket hub // Artifactory // ECR – git for images

Docker networking enables containers to connect to each other   
and to non-Docker workloads

1. Create a Network:  
 - docker network create  
2. Run/Connect running container in the Network  
 - docker run --network=  
 - docker network connect  
3. Inspect a Network to find details  
 - docker network inspect

Docker Swarm is a good choice for simpler applications that are quick to deploy and easy to manage.  
- Docker Swarm is easier to install and configure  
- Docker Swarm is known for its simplicity.   
In case if you’re already known with Docker commands.  
  
Kubernetes is better suited for complex, high-demand applications  
- Scalability features  
- Kubernetes has a larger community

Debug a running container

- Docker Logs  
- Docker Exec to execute commands like "container /bin/bash"  
- Docker Exec to get detailed information about your container

- Debugging Tools like an IntelliJ docker extension

JFrog's Artifactory

universal artifact repository manager.

Serves as a source of truth for all binaries and builds artifacts in the organization. Like security and access control, vulnerability analysis etc.

**Is highly useful in** **organizations** **having a single source of truth**

**Pros**: Integrates well with most CI/CD tools // Supports a wide range of package formats

**Cons**: advanced features come only with the enterprise plan // Complex initial setup

File Storing and Retrieval

Artifactory supports a variety of storage systems - local file system, on the cloud (AWS S3) or on a sharded file system

Duplication is avoided with the help of checksum even if the system is distributed

Retrieval: check is artifact presented in local cash and download it if not

**SDLC**

1. Set up Repositories

- **Local repositories:** these to deploy internal artifacts, e.g.,

- **Remote repositories:** These proxy and cache artifacts from remote resources

- **Virtual repositories:** combine local and remote repositories under a common URL to simple client-side configuration

2. **Configure Security:** Set up users, groups, and permissions

3. **Integrate with Build Tools:** configure it to resolve dependencies from and deploy build packages to Artifactory.

Versioning

**Repository Layout** – create layout with preferable dependencies

**Naming Convention**

**Version Control System versioning**

**Retention Policies**: Deploying

**Snapshots and Releases:** clean old and unused versions

Artifactory Query Language

flexible query language that allows you to fetch information about stored artifacts

**AQL is read-only**

Terraform

popular Infrastructure as Code (IaC) tool to create, modify, or destroy infrastructure using a declarative configuration language

**Steps:**

terraform init - sets up working directory

terraform plan - visualize changes

terraform apply - performs those changes

terraform destroy – to delete created

**Use when:**

- requirement is to manage a wide range of service providers

- unifying cloud service architectures

- repeatable infrastructure setup

**Pros:** - Cloud-agnostic approach

- easy-to-understand and self-document config files

- Ability to preview changes

**Cons**: Managing state (json file with changes) can be complicated, and resource dependencies can sometimes be difficult to handle.

- Unique syntax used

Architecture

Architecture

Monolithic Architecture

developed as a single, self-contained entity

+ Simpler to Develop  
+ Easier to Test  
+ Efficient Communication Between Components   
(you can speed up the process and improve performance)  
+ Single Deployment Unit  
+ Shared Memory Access

- Difficult to Maintain: (all the components are interlinked)  
- In big monolith needed understanding of the entire system. (Bus factor increases)  
- Limited Scalability  
- Slow Deployment Process  
- Changes Affect the Entire System  
- low Resilience: if one part of the application fails, the entire system can be affected

- app is relatively small, and the team is not large enough  
 - when app doesn’t require separate scaling  
 - when rush development is required

Microservices Architecture

breaks down an application into a collection of small services

+ Scalability   
+ Improved Fault Isolation  
+ Enhanced Team Productivity (decreased team communication)  
+ Quicker Deployment Time (of a single unit)  
+ Increased Cost-Efficiency (ability to use resources more effectively)

- Operational Complexity (you need to manage and orchestrate multiple services)  
 - Distributed System (need exactly know all business contracts around all upstreams and downstreams)  
 - Resource Consumption: (every single unit needs to have runtime environment,  
data storage or other resources)  
 - Management of Services (need of complex Orchestration Frameworks)

- application is large and complex  
- different teams are working on different parts  
- when you want to use different technologies for different services

Decomposition of microservices

- Domain-Driven Design (DDD): based on business capabilities or domains

- Decompose by Business Capability

- Decompose by Subdomain: different subdomains within your application

- Decompose by Process

- Decompose by Use Cases: Single Responsibility Principle

Serverless Architecture

applications are hosted by third-party service providers

+ No Server Management  
+ Cost-Efficiency (In perspective that code only runs when backend functions are needed)  
+ Automatic Scaling

- Cold Start Problems  
- Lock-In Concerns (high dependence on the ecosystem of other available services)  
- Debugging serverless applications can be challenging

- When the workload is unpredictable  
- When you want to reduce operational costs  
- When you want to benefit from potentially infinite scalability

Microservices patterns

By communication

**Event-Driven:** PECS with listeners (Lambdas)

**HTTP/API oriented:**

**Socket**: endpoint connection between two processes. It Can be **Stream-oriented** (guarantee the order and delivery) and **Datagram-oriented** (may arrive out of order or not at all)

Database Patterns:

**Database per service** isolates the data of each service

**Command Query Responsibility Segregation (CQRS):** separates the read and write operations per service.

**Sagas**: each transaction updates data within a single service.

Deployment Patterns:

**Single Service Instance per Host** (keeps services isolated from each other)

**Service Instance per Virtual Machine** (runs in its own virtual machine)

**Service Instance per Container** (easy way to manage and scale services)

**Serverless Deployment** (managed by the cloud service provider)

Visibility and understandability Patterns

**- health check APIs**

**- log aggregation**

**- distributed tracing**

**- auditing**

Strategies for handling fault   
tolerance and failover and for high availability

- Load Balancing (traffic balancing)  
- Auto-scaling (perf balancing)  
- Health Checks  
- Breaker Pattern: (works on Health Checks) stop calling failing downstreams  
- Replication  
- Transaction Management  
- Microservices Architecture  
- Container Orchestration: Kubernetes can automatically restart failed services

SQL vs NoSQL impact

SQL:  
- Scalability realization: SQL databases are typically scaled vertically   
- ACID realization: influence the design to ensure data consistency  
- Development Speed and Flexibility:   
 Changes in SQL database require altering the schema  
- Object Mapping realization: Object-Relational Mapping (ORM)  
  
NoSQL:  
- Scalability realization: NoSQL databases are designed to scale horizontally  
- ACID realization: handle eventual consistency and relax about acid rules  
- Development Speed and Flexibility:   
 can store different types of data in different ways,   
 So NoSQL is more flexible  
- Object Mapping realization: Object-Document Mapper

Main design elements

Architecture // Containerization // Orchestration // Stateless Design // Configuration and Secrets Management // CI/CD // Resilience // Security

Security measures

- Authentication and Authorization  
- API Gateway (that acts as a single-entry point into your system)  
- HTTPS  
- Service-to-Service Communication  
- Dependency Management  
- Secrets Management  
- Security Headers  
- Logging and Monitoring

Communication Type : refers to the method or mode of communication (Point-to-Point, Point-to-Multipoint)

Communication protocol refers set of rules of how data is transmitted and received (HTTP, HTTPS, TSP)

Protocols

- **HTTP** request-response protocol

- **HTTPS** secure version of HTTP uses SSL/TLS protocol to encrypt the data communication (exchange of digital certificates and shared secret key to encrypt the data)

- **FTP** transferring files

- **TCP / UDP** connection-oriented protocols. UDP does not guarantee delivery but faster

- **SMTP / IMAP / POP3** mail delivery

- **WebSocket** full-duplex communication channels over a single TCP

Programming Architecture

Compiled languages

are translated into machine code, which can be executed by the processor

+ They run faster and more efficiently  
+ Has better control over hardware resources  
  
- require an additional ‘build’ stage  
- Every time you make changes, you will need to ‘rebuild’ the program

Interpreted languages

are read and executed line by line by another program

+ immediately sees all changes and translates it to you  
+ JIT (Just-In-Time) compilation  
 + dynamic recompilation   
 + microarchitecture-specific speedups  
  
- Interpreted languages are much slower than compiled ones.

Is java compiled or interpreted

Compilation: Java source code is first compiled into bytecode by the Java compiler.  
  
Interpretation: The JVM interprets and executes this bytecode at runtime.  
  
Just-In-Time Compilation: Modern JVMs also have a Just-In-Time (JIT) compiler.

JDK, SDK, JRE

**SDK** (Software Development Kit) - tools to develop applications (can include JDK)

**JDK** (Java Development Kit) - for developing Java apps (include JRE)

**JRE** (Java Runtime Environment) - components run a Java apps (JVM, libs)

Programming paradigms

Soft Proc

SDLC steps

1. Analysis and planning 2. Requirements 3. Design and prototyping

4. Software development 5. Testing

6. Deployment 7. Maintenance and updates

SDLC Methodologies

**Agile**: focuses on iterative progress and flexibility to changes.

**Waterfall**: linear approach where each phase completed before starting the next one.

**Rapid Application Development (RAD):** uses minimal planning and rapid prototyping for quick development

**Feature-Driven Development (FDD):** short-iteration process of feature development

**TDD (Test-Driven Development)** - where the tests are written first, then the code is developed to pass those tests.

**BDD (Behavior-Driven Development)** – behavior is specified in a common language that can be understood by all involved (example: Cucumber)

**ATDD (Acceptance Test-Driven Development)** - tests are created even before the code is written but these tests are created with the collaboration of stakeholders, developers, and testers.

SDLC Tools

**Logging** – log4j / Cloud Logging / Splunk / Console logging

**Tracking** – Debug / JVM review / DataDog

**Monitoring** – Java VisualVM / JProfiler / LightStep / Pinpoint / MosKito / BlackDuck / DataDog / Amazon X-Ray / CloudWatch

Error resolution scenario

- **Error Identification**: (checking the error logs, monitoring systems, or user reports)

- **Replicate the Error**: (understand the conditions under which the error occurs)

- **Analyze the Error**: to understand what is causing the error

- **Deep investigation**

- **Fix the Error / - Test / - Deploy**

- **Document / Agile Retro**, what we can do not to have this problem in future

Steps to choose software solutions

- Define Your Requirements (performance, security, and usability)

- Research Available Solutions

(commercial, open-source or custom-build)

- Think about Each Solution (features, ease of use, scalability, reliability, and cost)

- Test the Software (under conditions that are as close as possible)

- Check Compatibility

- Make a Decision

- Implement the Software

Patterns

General patterns

**Builder** - solves the problem of creating objects with many parameters, without creating large constructors for all cases, and without creating many subclasses.

**Decorator** - in the case when we have several implementations, but we need to have create combo implementations. Create a layer between the interface and implementations and make kombo calls.

**Facade** - simple interface to a complex subsystem. As an example, call center. This is one simple call method, and it already calls complex methods for processing.

**Adapter** - allows objects with mismatched interfaces to work together. For example, the response of one service in xml and the input of another in json. Create an additional data mapper layer and create a json based on xml

SOLID

**Single-Responsibility** – one class one responsibility. simplification of code work, less merge conflicts, avoid of creation god classes that are difficult to update

**Open-Closed** - open for extension and closed to modification. we should add new behavior without affecting existing code. This is because if we modify existing code, we have a bug risk.

**Liskov Substitution** - Describes the relationships between base and inherit classes. Subclasses should be substitutable for their base classes.  
 - **Subtype Requirement** if app is using a base class, it should be able to use any subclasses

- **Method Signature** Subclasses must follow base class method signature

- **No New Exceptions** Subclasses should not throw exceptions that are not thrown by the methods of the superclass

**Interface Segregation** - many client-specific interfaces are better than one general-purpose interface

**Dependency Inversion** - classes should depend on abstraction not of concrete classes.

**DRY** - Don't Repeat Yourself - do not use the same block of code in the program but use a call to this block of code.

Code Quality

Clean code principles

- Meaningful Names   
- Functions Should Do One Thing   
- Short Functions (5,50,500 rule)   
- Function Arguments (less as possible)   
- No Side Effects  
- Use Exceptions handles  
- All changeable data must be kept on High Level  
- Do Not Use Magic Numbers or Strings

- Documentation and self-explain code  
- Code Formatting

Code Refactoring principles

- Method decomposition to avoid god classes

- Method composition to avoid code duplication

- Put hard expression result into self-descriptive local variables

- Hard expression decomposition for code readability

- Replace temporary vars to expressions for better maintainability

- Rename magic vars and methods

- SOLID, DRY, KISS, Patterns

SonarQube

open-source platform used to measure and analyze the quality of source code

**Pros**: support for a wide range of programming languages

- Easy integration with code and devOps tools

- provides useful and actionable metrics

**Cons**: Can be tricky to configure to suit specific needs

- can be resource-heavy

- can produce false positives

Security

General Security

Secure the Source Code  
- Static Code Analysis  
- Dependency Check  
- Regularly Update Dependencies  
  
Secure the Build Process  
- Secrets Management  
- Build Isolation  
- Automated Security Testing  
  
Secure the Deployment  
- Immutable Infrastructure  
- Least Privilege Principle  
  
Monitor and Respond  
- Audit Logs

Defense in depth is a cyber-security strategy that using multiple layers of security controls if one layer of defense turns out to be inadequate, another layer of defense will hopefully prevent a full breach.

- Layered Security (network firewalls, data encryption, secure coding practices)

- Principle of Least Privilege

- Regular Updates and Patches (fix known vulnerabilities)

- Use of Security Tools (intrusion detection systems, security scanners)

- Regular Audits

- Employee Training

- Incident Response Plan

- Data Backups

Authentication and Authorization

**Authentication**: verifying the identity of a user, device, or system.

- Strong Password Policies

- Multi-Factor Authentication

- Limit Login Attempts (prevent brute force)

- Use Secure Protocols

**Authorization:** authentication + determination allowed policies of user

- Principle of Least Privilege

- Role-Based Access Control

- Regular Audits

- Use of Access Control Lists

Rest security

**Transit Level**

- Use Secure Protocols (HTTPS for web, SFTP for file, TLS for email)

- SSL/TLS Certificates to provide encrypted connection

- VPN

**Rest level**

- Encryption // - Access Controls // - Regular Audits // - Secure Backups

**Both**

API Key Management that used to authenticate a user and control API usage (do not provide protection against attack) (generation, storage, rotation)

// Data Minimization (minimum amount of sensitive data)

Session management

**Session ID:** (long enough to prevent brute force attacks)

**Secure Transmission**

**Regenerate Session ID** // **Timeout** // **Logout** // **Session Termination**

**Secure Cookie Attributes**

**Server-Side Session Management**

Security headers

- HTTP Strict Transport Security (header tells the browser to always use HTTPS)

- Content Security Policy (CSP): prevent cross-site scripting (XSS)

- X-Content-Type-Options:

- X-Frame-Options:

- X-XSS-Protection

- Access-Control-Allow-Origin: (which websites can make cross-origin requests)

Security frameworks and protocols

OAuth 2.0

authorization framework that enables to have **secure access to all services at ones using HTTP protocol**. (login to google maps will login to google cloud)

pros: login to al services at ones

- Request authorization from the server.

- Get authorization grant from server.

- Request an access token.

- Get an access token.

- Present the access token to get any resource.

- If the access token is valid, the resource server serves the resource to the client.

**Pros**: Delegated Access // Flexibility

**Cons**: can be trickier to manage and revoke access to resources // Incompatibility with Previous Version

OpenID

allows users to authenticate across multiple websites using third-party services, eliminating the need to create separate account

**pros**: simplified account management

**cons**: dependency on third-party services

steps:

- login redirection to third-party

- provider generates a response that includes whether authentication was successful

- Site verifies the response from provider and authenticates user

Security Assertion Markup Language (SAML / SAML 2)

SAML (**Service Provider**) is standard for **exchanging authentication and authorization data**

SAML2 (**Identity Provider**), an extension of SAML, is an XML-based protocol. Used to **pass information about a principal** (usually an end user) **from a SAML**.

General use: Providing secure access to apps and services across different domains

When implementing single sign-on

**Pros:** Supports both frontand back communication (browser-server and server-2-server)  
**SAML avoids password sharing**

**Cons:** XML-based protocol, can ne complex

JSON Web Token (JWT)

open standard used to share security information between client and service.

Used for authentication and authorization purposes.

- A client sends a login request

- The server verifies the credentials. Generates a JWT and sends it back.

- The client stores the JWT, often in local storage or a cookie, and includes it in the header.

- The server verifies the JWT signature to processes the request or return error

Benefits:

+ Stateless, Scalable Authentication

+ Fine-Grained Access Control

+ Mobile-Friendly

OAuth and JWT used together, OAuth 2.0 for authorize the application and JWT to represent the user's identity and permissions.

How to detect security vulnerabilities

OWASP Zed Attack Proxy (ZAP)

free security tool for automatically finding security vulnerabilities. Passing raw traffic into application.

- Allow finding vulnerabilities during development

- passive and active scanning of security vulnerabilities

**Pros**: highly configurable // open-source

**Cons**: require security expertise (need to know what to find)

- scans can sometimes be time-consuming

OWASP Dependency Check

open-source tool to detect publicly disclosed vulnerabilities

checking project dependencies in Common Vulnerabilities and Exposures (CVE) database.

**Pros:** Can be used as a CI /CD tool

Can generate reports in multiple formats

**Cons:** It may return false positives, (requiring manual triage)

Performance might be slow

Frontend tech stack

Basics

**Cookie**: small piece of data stored on the client's computer

**Session**: information on the server-side collected during a series of requests

JavaScript

**Synchronous vs Asynchronous**

Synchronous:  
- Latency: can lead to increased latency  
- Complexity: generally easier to implement  
- Resource: increased resource cost  
- Error Handling: easy to handle  
- Ordering: guarantee the order of messages  
  
Asynchronous:  
- Latency: reducing latency  
- Complexity: can be more complex to implement correctly  
- Resources: better resource optimization  
- Error Handling: can be more complex  
- Ordering: does not guarantee the order of messages

LTS updates

Optional Chaining – get value of a property located in a chain without check of all chain for not null

Nullish ternary – logic condition that returns left value if right is null and so on

Angular

LTS updates

Versions 10 and 11 now are in LTS

- Faster Builds

- Automatic Inlining of Fonts

- Improved Reporting and Logging

TypeScript

LTS updates (4.0)

- Speed Improvements

- Log improvements

- added variated unknown tuple types and support for them like high level operations and labels

Another Techs

Kotlin

is an Object-oriented and Functional programming language developed by JetBrains.   
  
The main idea behind the creation of Kotlin was to develop a new language   
for the JVM (Java Virtual Machine) that would be more efficient and productive than Java

+ compile faster than java and save java Multiplatform Capability  
+ 100% associated with Java in both directions  
+ designed to be more readable and simpler to use as Java

+ Null Safety improvements

- all disadvantages of young language  
 - small community  
 - not reach resources for learning  
- in some area’s compilation speed can be slower  
- still, we have association issues with some java libs

var (Variable): It is a mutable variable, value can be changed anytime   
  
val (Value): It is an immutable variable, similar to a final variable in Java

How does null safety work?

**Nullable and Non-Nullable Types**: in Kotlin all types are non-null by design.  
But I can create nullable value String?  
**Null Checks:** When you want to access a nullable reference, you must handle the null case check

**Safe Calls** Kotlin provides a safe call operator **(?.)**  
**Elvis Operator:** **(?:)** to return value in case of null  
**Not-null Assertion Operator (!!)** (exclamation mart)

coroutines are a design pattern that asynchronously runs the code. Main idea is the programming on Android to manage long-running tasks and not to block main thread

Data class vs Regular classes

**By use case:** D used to hold data. R data and behaviors

D cannot be extended by another class

D cannot be sealed, open, abstract, or inner

D automatically generate equals(), hashCode() based on prim constructor

companion object like static

lambda

in Kotlin this is anonymous functions:

val multiply = { a: Int, b: Int -> a \* b }

also, this can be function param

fun operateOnNumbers(a: Int, b: Int, operation: (Int, Int) -> Int): Int {}

…

operateOnNumbers(5, 2, { a, b -> a \* b })

Extension functions allow you to extend a class with new functionality without having to inherit

When expression is switch case

== for Structural Equality

=== for memory refer equality

- With: used for simple object clone (return some other object) (easy clone)  
- Let: used in null check/non obj state change operations  
- Apply: used for object configuration and returning the object itself  
- Run: used for executing a block of code on an object   
and returning the result of the block.  
- Also: used for performing additional operations or side effects  
on an object and returning the object itself.

(the whole group is syntactic sugar to simplify the code)

Soft skills

Main responsibilities

- Coding and Development

- Testing and Debugging

- Collaboration and Teamwork

- Code Review / Documentation

Enjoyable aspects of work

- Problem-Solving

- Continuous Learning

- Team Collaboration

- Innovation / Flexibility

- Impact

What can be improved

- Knowledge of new technologies

- Code optimization

- Communication skills

- Testing and Debugging

- Architectural Design

- Understanding the business domain

Meeting Work Commitments

- personal deadlines

- making daily to-do lists

- prioritizing tasks

- using project management tools

- team collaboration