Java Core

**OOP**

Encapsulation – relationship

Inheritance – expand

Polymorphism – same method in diff ways

Abstraction – get only needed params

**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

**Meta** annotations that extend the behavior of already created annotations.

@Target @Retention

(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

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)

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

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

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.

**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”

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

(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

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

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** 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

SQL

DBMS (Database Management System)

**Relational** – mane objects presented as tables with PK and FK (Azure Data Explorer, MySQL Query Analyzer)

**Non-relational** – mane objects presented as not only tables/json/docs/xml

**Normalization**

**1NF –** no duplicates and use of PK

**2NF** - concept of removing duplicative data. Non-PK columns can’t depend on each other

**3NF** - All columns must depend of PK

**ACID**

**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

**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 (work with different data)

**Repeatable read** - transactions can block other rows for writing or reading (phantom read)

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

**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

**Indexes** are special tables that a database search engine can use to speed up data search.

**Single column // Unique // Composite** // **Default**

REST

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

**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. + HATEOS + selfDescriptive

**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)

**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

Spring

+ many components

+ AOP and easy MVC

+ implementation of IoC

- high entry threshold

- may be slower in some situations as concurrent

- this is framework

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

Singleton – like pattern (control by Sprig)

Prototype – single object (creation control

Request – bean scoped to HTTP req

Session – http session

G-session – global http session

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

Bean LC

- Obj Constructor

- Dependency injection \*Autowiring

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

- initialization step (@PostCostruct/initMethod/afterPropertiesSet ())

- post init (BeanPostProssesor.postProcAfterInit())

Destroy LC

- @PreDestroy

- destroyMethod

- DisposableBean.destroy()

@Autowired (fields, method or m init params, constructor) @Quilifier @Lazy @Bean

Configure

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

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 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 AOP and AspectJ logically separated

Spring AOP gives us common AOP solution for beans

AspectJ gives us complete AOP solution

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

**Waterfall**

Complete design and planning

In case of changes, you need to return to the previous steps, which increases costs and time.

**Agile and Scrum**

You can make changes to the product at any time

Uncertainty. The number of sprints can be unlimited, you need to clearly select a team

Product – Spring backlogs/ Sprint review/ Development / Feedback / Retro / morning scrum

**Kanban**

Kanban is a methodology whose main objective is to visualize the process, and Scrum is built around the team.

**Incremental and Iterative**

Planning - Implementation - Verification - Evaluation

early creation of working software

problems with the implementation of the overall system architecture

Logging – log4j / Cloud Logging

Tracking – Debug/ JVM review

Monitoring – Java VisualVM/ JProfiler/ LightStep/ Pinpoint/ MosKito/BlackDuck

Amazon X-Ray/ CloudWatch

The point of CI is quick feedback for developers.

- execution of tests; - dev envs deployment – sending reports

Continuous Deployment is the practice of automating the entire software release process. Automatically prepare and push

the release to production.

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

**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** - subclasses should be substitutable for their base classes.

**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.

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

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

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

AWS

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

Basics

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

**AMI** (Amazon Machine Images)

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

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

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

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

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

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)

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 recover 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

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)