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

CI/CD with Docker

Build Docker Images  
Run Tests in Docker Containers  
Push Docker Images to a Registry  
Deploy Docker Images  
  
Pluses:  
Consistency — we have the same env during development  
Isolation — containers are isolated from each other  
Scalability — can be easily scaled up or down  
Portability — run on any system  
  
Minuses:  
Complexity of knowledge and integration: can be a problem for who are new to Docker  
Security - Docker containers are isolated from each other, but still need security  
Framework Dependency

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 Achitecture

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.

Docker

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

**String** objects are themselves immutable

**StringBuffer** is synchronized

**StringBuilder** is faster

**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

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

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

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

NEW /RUNNABLE /BLOCKED /TIME\_W /TERMINATED

Manual if else Interrupt() deprecated stop()

yield() says another thread to run

join () say I will work when you end

Happens before – JVM can change instructions. Volatile to solve problem

PECS – some threads add values to blocking queue, some get values

ReentrantLock – get thread permission to enter the monitor without stopping (lock, tryLock, unlock)

ReadWriteLock – block reading or writing

IO – r/w at once. Thread saves

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

Serializable – classic way

Externalizable - read/writeExternal

If parent is ser. Childs also

If field is referenced need to be ser.

Serial \ Parallel (default) \ CMS \ G1

-------------------------------------------------------

**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

**Stream** – code simplification with building call chains. (peak)

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

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

-------------------------------------------------------

**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)

-------------------------------------------------------

**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.

**Java versions**

Java 9 -16

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

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

**AWS**

DB: AuroraDB / DinamoDB

CI/CD: code pipeline/ code deploy/ code build / code commit.

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

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

AMI (Amazon Machine Images)

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

Memcached: for simple cashing

Redis: for more functions

Elastic Load Balancer: helps balance traffic between several downstream instances

Identity and Access Management (IAM):provides securely control

Lambdas**:** event driving processors

X-Ray**:** Debugging analyzing service

CloudWatch**:** monitoring service

CloudTrail**:** monitoring user activity

CloudFront**:** speeds up distribution

of your static and dynamic web content

Web Access Firewall: protect against common network exploits

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

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.

Athena: interactive query service

Kinesis: collect process and

analyze streaming data in real time

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

CloudWatch Evidently: feature for performance monitoring

Macie: AI data security service

Fargate: service to run containerized applications

Systems Manager Parameter Store: for system parameters

Secrets Manager with KMS: for secure generation and containing secrets

Simple Notification Service: notify all subscribers

Simple Queue Service: containing messages

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

**Primitives | Expand | Reflections api | cheap | expensive |**

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