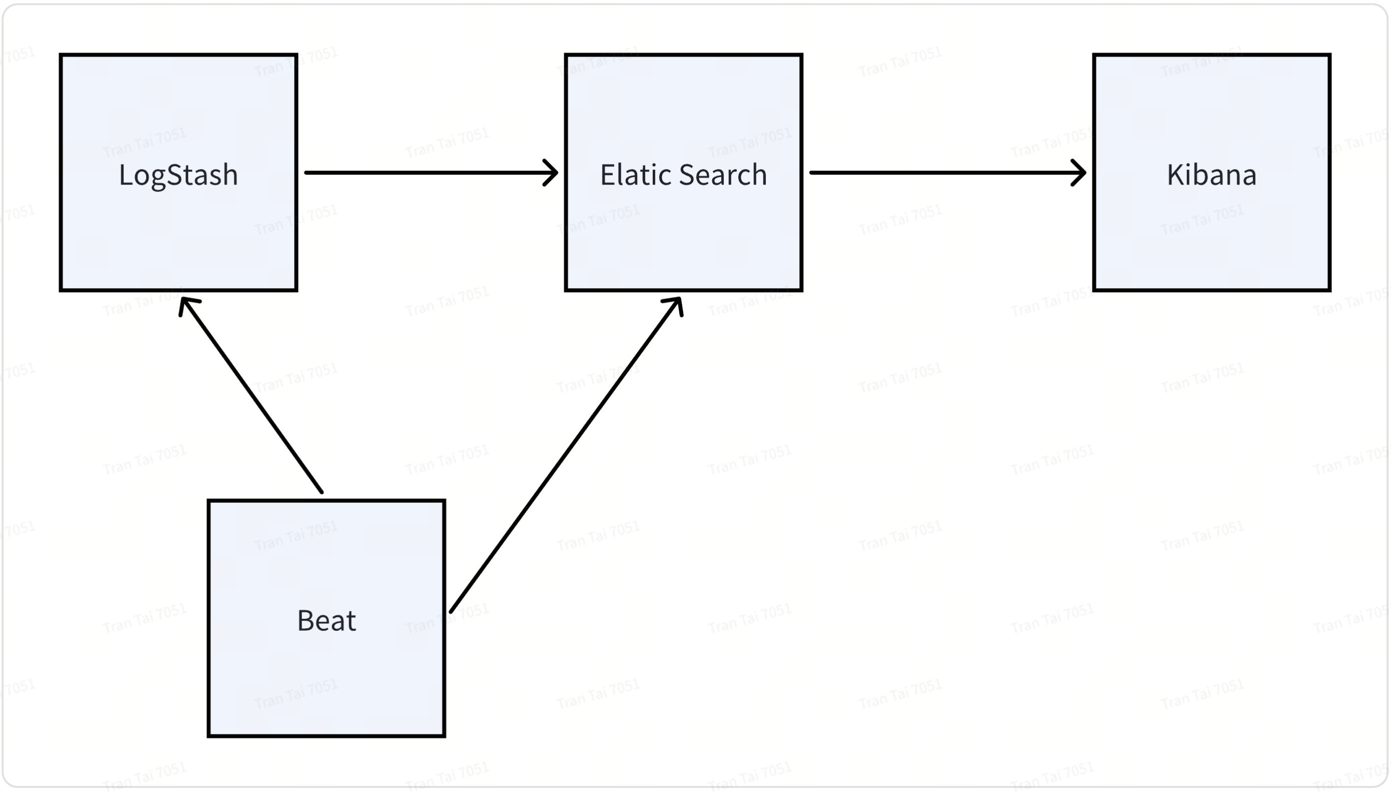
Noted Questions

Case study designs

How ELK works?



- ELK (Elastic Search - LogStash - Kibana)
- LogStash/ Beat: ingest data in elastic search.
- Kibana: provides interface for the user to construct the filter. It will then construct a RESTFUL api call to Elastic search and convert the user filter into a DSL query send to elastic search.

All message queues stuffs? When using different types of MQ (RabbitMQ, RedisMQ, Kafka)

Connection Pool design.

Elastic Search

Kafka

Spatial Indexing Tree

- Quad Tree
- R Tree
- KD Tree

Consensus Algorithm

Different synchronization primatives and when used? Lock, Channel, Semaphore?

- Memory type in OS:
  - Virtual Memory/Page fault.
  - Scheduling algorithms.

System design.

  - Try to solve the problem
  - Watching Engineerpro vid + Bytebytego.
  - Upsolve Alex Xu book.

OS

Thread/ Process

Lock/Semaphore/Conditional variable

Context Switch

Thread/Process Communication

- Message Passing
- Shared Memory

Network (TCP/UDP/Congestion)

TCP/UDP differences.

Congestion control algo.

HTTP 1.0 and HTTP 2.0

TLS termination

TCP handshake

Programming Language (Golang/C++/Java)

Java

Java program -(compile)-> bytecode -(run)-> JVM

- Bytecode is **independent of the software or hardware running.** JVM is platform dependent but as bytecode can be executed in any other system/JVM makes Java platform independent. JVM is an **interpreter** for loading/verifying/executing java bytecode.

- 4 pillars of OODPs:
  - Abstraction
  - Inheritance
  - Encapsulation
  - Polymophism

- Thread Pool
- **Blocking queue** and **Golang channel** differences

    - SELECT capability is hard to reproduce with blocking queue.

- How GC works?
    - Collector **scans the heap** looking for objects that are no longer in use.
    - If the object no longer has any ref, collector **removes the object**, **frees up memory in heap**.
    - JVM divide heap into separate spaces and GC use mark-and-sweep algo to traverse the space and clear out.
    - Different kinds of heap spaces.
        - **Eden space**: most objects when first appear in heap will be put here.
        - **Survivor space**: there are 2 survivor spaces (one and zero). It is also a part of **young generation** heap memory.
        - **Tenured space**: Long-lived object are stored aft they have passed some cycles of GC.

        From top to bottom, the GC scan frequency will be less active for efficiency as the younger space, the higher chance that the object lifecycle there will be shorter compared to older space. Need some balance for the GC scan frequency in tenured space to avoid memory leaks.

        - **Permanent generation**: for storing required application metadata for JVM. (already be removed in Java 8)

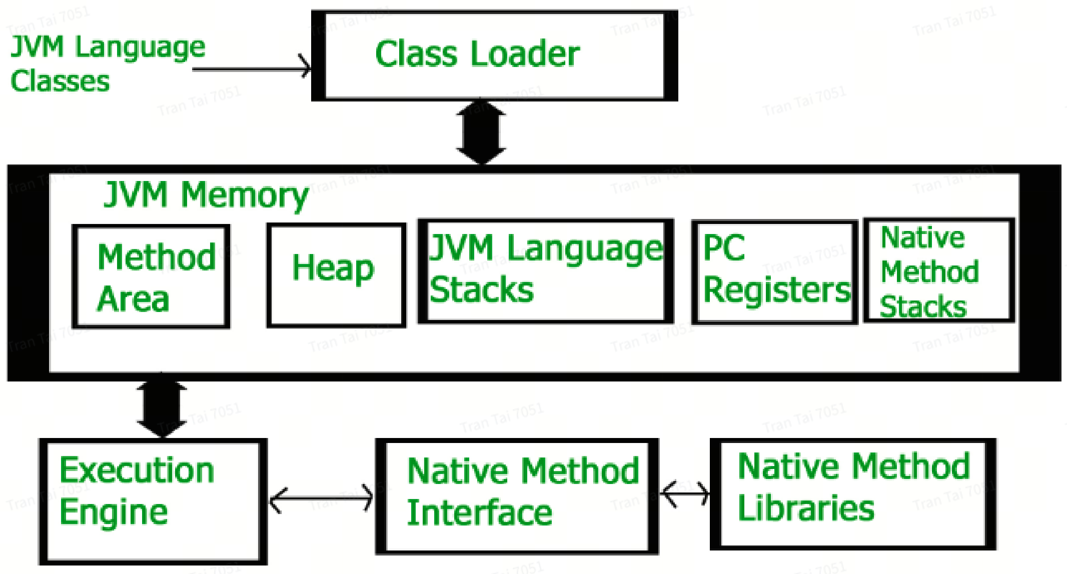
    - Mark and Sweep algo:
        - When an object created in heap -> a bit mark to be zero.
        - Mark phase: GC traverse from the roots and mark all reachable object from root to 1. Mark bit for unreachable is unchanged.
        - Sweep phase: GC traverse the heap, reclaim mem from all items with mark bit = 0.
        - The GC will be triggerred in different kind of events:
            - Minor event: eden spaces is full, objects moved to survivor spaces.
            - Mixed event: minor events reclaim old generation objects.
            - Major event: clear space in both young and old gen.

- Different type of GCs strategies:
  - Serial
    - Aka "stop the world"
    - Use for small and single threaded env.
    - When GC is run, stop action of all other threads/processes.
  - Parallel.
    - JVM Default GC strategy.
    - Utilize multiple threads for cleaning up.
    - Also interrupt/freeze other threads.
  - Concurrent Mark and Sweep (CMS).
    - Similar with parallel strategy by using multiple threads.
    - Low pause by freezing the application less frequently.
    - Only can concurrently collect old generation mem, still need to freeze the application when collecting young generation. **WHY?**
    - GC thread works at the same time as application thread -> more resource intensive.
  - Garbage first (G1)
    - Mixed up approaches tgt.
    - Divide the heap in many spaces. (not only eden/survivor/tenure) and clean the minimum space first.
    - Run concurrently like CMS but less freeze and can collect young/old generation concurrently.

- Java OODPs

How JVM works

- A runtime engine to run Java application.
- Part of JRE (Java Runtime Environment).



Class Loader Subsystem:

- Responsible for **Loading**/**Linker**/**Initialization**.
  - **Loading**
    - **Class loader** read .class file -> binary data -> save in **method area**
    - For each .class file, JVM stores those infos in method area.
      - Fully qualified name of loaded class and its immediate parent class.
      - .Class file related to Class/Interface/Enum.
      - Modifier/Variables/Method info.
    - Aft loading .class, JVM creates **object of type class** to represent the file in **heap memory**. These class objects can be used to get class level info like class names, parent names, methods, variable infos.
  - **Linking**
    - **Verification**: ensure correctness of .class file (valid format, generated by valid compiler). If fail -> received run-time exception error. This is done by ByteCodeVerifier. Once this is completed -> class file is ready for compilation.
    - **Preparation**: allocates memory for **class static variables** and init memory to default values.
    - (Optional) **Resolution**: replace symbolic reference from the type with direct reference. This required searching into method area to locate referenced entity.
  - **Initialization**

        - **All static variables assigned with values in code** and static block. This execute from top to bottom in a class and from parent to child in class hierarchy.
    - In general there are 3 class loaders:
        - **Bootstrap**: loading trusted classes (e.g. Core java api classes)
        - **Extension**: child of bootstrap class loader. Load classes present in the extensions *JAVA_HOME/jre/lib/ext"* (Extension path) or any other directory specified by the java.ext.dirs system directories.
        - **System/Application:** child of extension class loader. Load class from application classpath. Internally use environment variables which are mapped to java.class.path.

JVM memory

- **Method Area**: all class level info stored here (class name, variables, methods). One method area per JVM and it is a shared resource.
- **Heap Area:** info of all objects stored here. Only one heap area per JVM and it is a shared resource.
- **Stack Area**: for every thread, JVM creates one run-time stack. Every block of the stack called activation record/stack frame which store methods calls. All local variables stored in corresponding frame. Aft thread is terminate, the runtime stack will be destroyed by JVM. This is not a shared resource.
- **PC registers**: Address of current instruction of a thread. Each thread has separate PC Registers.
- **Native Method Stacks**: similar with stack except it stores native method info.

Execute Engine

- Execute .class (bytecode) by reading bytecode line-by-line, use data and info in various memory areas and execute instructions.
- 3 parts:
    - **Interpreter**: interpret the bytecode line by line and execute.
    - **JIT compiler**: increase efficiency of interpreter by compiling entire bytecode -> change to native code, so when repeated method is calling again, JIT provides direct native code for that part.
    - **GC**: destroyed unreferenced objects.

Java Native Interface

- Interface that interacts with Native Method Libraries and provides native libraries (C/C++) required for execution. It enabled JVM to call C/C++ and to be called by C/C++ libraries which may be specific to hardware.

Native Method Libraries

- Collection of native libs (C/C++) which are required by execution engines.

- Notify, wait ,sync block

- Java Spring overview

C++

- alloc/malloc/dealloc.
- Std library.

Golang

- Goroutine
- Work stealing
- Concurrency: Design Patterns + Primatives

Containerization/ Virtualization

Docker

K8s