1. Database:
    a. SQL vs NoSQL uses cases
    b. ACID vs CAP
    c. NoSQL types and use cases
    d. Partitioning vs Sharding
    e. Indexing (B-Trees etc)
2. Exception Handling
    a. Which layer (architecture wise)
    b. Language specific
    c. Microservices world
3. Microservices
    a. General idea
    b. Pros and cons
    c. Use case
    d. Monolith -> Microservice - decisions, when, how, tradeoffs
4. Scaling
    a. Horizontal vs Vertical
    b. Application level vs DB level - use cases
    c. Application level - concurrency vs parallelism, queueing (Push, Pull based approach), caching (internal vs external)
    d. Concurrency - short (multithreading), long async tasks (queue), Concurrency Models
    e. DB level - CAP and ACID - tradeoffs - scaling reads and writes independently n how
5. DS and Algo
    a. Simple ds
    b. Simple algo
6. Languages
    a. Exceptions
    b. Single vs multi threading support
    c. Basic concepts
    d. Strongly typed vs loosely typed (comparison, tradeoffs)
    e. Thread level debugging in java
7. Frameworks
    a. Spring - use cases, features, tradeoffs
    b. MVC
    c. AOP
    d. DI - why, what?
8. Distributed Systems
    a. Idea
    b. Databases internal working/architecture (mongo, dynamo)
    c. Architecture - Master master vs Master slave - tradeoffs
    d. CAP theorem
    e. Gossip protocol / Ring membership protocol
    f. Consistent Hashing
9. Functional Paradigm

    a. Immutability
    b. Pure functions
    c. Lazy evaluation
    d. Type Systems
    e. Monads, functors, Applicatives
10. Event Driven Architecture
    a. SAGA pattern
    b. CQRS
    c. Distributed Transactions
11. HTTP and Web Socket
    a. How browser works? DOM model, V8 engine etc.
    b. Http 1.1 and 2.0
    c. WS internal working on UDP and dependency on HTTP for handshake
    d. WS use case, why not Http everywhere
12. JavaScript
    a. Basic Objects, functions, Arrays
    b. Node.js - internal working, async, single threaded
    c. Functional programming ES6
13. Big Data
    a. Frameworks
    b. Ecosystem
    c. Streaming data
    d. Recommendation systems
    e. Administration/Managing Clusters
14. System Design
    a. Architecture
    b. Components - what, why?
    c. Database choice - sql / Nosql
    d. Services or APIs
15. OO Design
    a. Real world representation
    b. Behaviour and responsibility
    c. SOLID principles
16. Design Pattern
    a. Factory, Builder, Strategy and few other commonly used patterns
17. Crypto
    a. Blockchain basics
    b. Cryptography - public/private keys
    c. Merkle Trees
    d. Bloom Filters

RDBMS (SQL) vs NoSQL

queries
- sql - wins here with natural query language
- nosql
  - supports aggregates using Map-Reduce
  - limited at joins

transactions
- sql - supports across rows
- nosql
  - most of them supports at single row level
  - google cloud store supports at entity group level as well as cross entity group level
    - involves 2 phase commit so this doesn't scale well & have limitation of 5 entity groups
  - mongodb supports at single document level

consistency
- sql
  - strong consistency if only one node is involved
  - for multiple nodes - mostly involves master-slave architecture and async replication which may cause data loss if master fails before complete replication and sync replication can cause latency
- nosql
  - mostly involves peer-to-peer architecture with parallel transaction log replication, but this causes eventual consistency

scalability
- sql
  - vertical scaling mostly
  - some of them supports sharding but not that efficient
- nosql
  - scales linearly for thousands of requests - achieved by smart data partitioning with load balancing
  - horizontal scaling
  - easily supports large volumes

data modeling
- sql
  - everything ahead of time
  - design schema, relationships and constraints - strictly enforced
  - best practice is design highly optimized schema so that data redundancy can be avoided (write optimized)
  - schema design

  - supports ALTER table and column operations - inefficient (may involve locking the whole table)
  - some of the databases support online schema change - background copy and rename

- nosql

- no need to define the schema ahead of time
- model the data according to the need of application - read/write optimized
- no table level joins supported
- most of the do not support row/table level constraints

===

RDBMS
- lot of querying on each and every field with joins n all
- transactions on whole data
- strong consistency for all the operations
- strict schema enforcement

NoSQL
- requires scalability for large volumes and velocity
- schema changes

RDBMS

pros:
- Persistence
- SQL
- Transactions (concurrency management)
- reporting
- Relationships are represented by data.
- Reduced duplication of data in database can be achieved by normalization.
- They allow greater flexibility and efficiency for structured data

cons:
- impedance mismatch - one logical structure in application maps to 10 tables in database
- scaling
  - have practical limits
    - vertical - cost limits and size
    - horizontal - ACID & latency limits
- handle unstructured data which is non-relational and schema-less in nature
  - overhead of joins and maintaining relationships amongst various data

NoSQL

pros:
- High scalability
- Distributed Computing

- Lower cost
- Schema flexibility
- Un/semi-structured data
- No complex relationships - so no joins or constraints at db level

RDBMS Scaling

master-slave replication
master-master replication
Federation
Sharding
Denormalization
SQL tuning

https://github.com/donnemartin/system-design-primer#relational-database-management-system-rdbms

Why NoSQL stands high on scalability??

Scale has to be broken down into its constituents:

Read scaling = handle higher volumes of read operations
Write scaling = handle higher volumes of write operations

ACID-compliant databases (like traditional RDBMS's) can scale reads. They are not inherently less efficient than NoSQL databases because the (possible) performance bottlenecks are introduced by things NoSQL (sometimes) lacks (like joins and where restrictions) which you can opt not to use. Clustered SQL **RDBMS's can scale reads by introducing additional nodes** in the cluster. There are constraints to how far read operations can be scaled, but these are imposed by the difficulty of scaling up writes as you introduce more nodes into the cluster.

Write scaling is where things get hairy. There are various constraints imposed by the ACID principle which you do not see in eventually-consistent (BASE) architectures:

Atomicity means that transactions must complete or fail as a whole, so a lot of ***bookkeeping*** must be done behind the scenes to guarantee this.
Consistency constraints mean that all nodes in the cluster must be identical. If you write to one node, this ***write must be copied to all other nodes*** before returning a response to the client. This makes a traditional RDBMS cluster hard to scale.
Durability constraints mean that in order to never lose a write you must ensure that before a response is returned to the client, the ***write has been flushed to disk***.
To scale up write operations or the number of nodes in a cluster beyond a certain point you have to be able to relax some of the ACID requirements:

Dropping **Atomicity** lets you shorten the duration for which tables (sets of data) are ***locked***. Example: MongoDB, CouchDB.
Dropping **Consistency** lets you ***scale up writes across cluster*** nodes. Examples: riak, cassandra.
Dropping **Durability** lets you respond to write commands ***without flushing to disk***. Examples: memcache, redis.
NoSQL databases typically follow the BASE model instead of the ACID model. They ***give up the A, C and/or D requirements, and in return they improve scalability.*** Some, like Cassandra, let you opt into ACID's guarantees when you need them. However, not all NoSQL databases are more scalable all the time.

The SQL API lacks a mechanism to describe queries where ACID's requirements are relaxed. This is why the BASE databases are all NoSQL.

Most cases where NoSQL is currently being used to improve performance, a solution would be possible on a proper RDBMS by using a correctly normalized schema with proper indexes. RDBMS's can scale to high workloads, if you use them appropriately.

Partitioning vs Sharding

Horizontal partitioning ***splits one or more tables by row, usually within a single instance of a schema and a database server***. It may offer an advantage by reducing index size (and thus search effort) provided that there is some obvious, robust, implicit way to identify in which table a particular row will be found, without first needing to search the index, e.g., the classic example of the 'CustomersEast' and 'CustomersWest' tables, where their zip code already indicates where they will be found.

Sharding goes beyond this: it partitions the problematic table(s) in the same way, but it does this ***across potentially multiple instances of the schema***. The obvious advantage would be that search load for the large partitioned table can now be split across multiple servers (logical or physical), not just multiple indexes on the same logical server.

There is also **Vertical Partitioning** (VP) whereby you ***split a table into smaller distinct parts***. Normalization also involves this splitting of columns across tables, but vertical partitioning goes beyond that and partitions columns even when already normalized.

MySQL sharding challenges:

- not directly supported by the database
- Application Queries Need to Route to Correct the Shard(s)
- Cross-Node Transactions and ACID Transactionality
- By not having a single RDBMS managing ACID transactionality across the shards, this functionality either has to be avoided (limiting workload and business flexibility), or has to be (re)built at the application level (at high cost and potential data risk).

https://www.quora.com/How-would-you-compare-MySQL-sharding-vs-Cassandra-vs-MongoDB

Horizontal vs Vertical Scaling

Horizontal scaling means that you scale by ***adding more machines into your pool*** of resources whereas **Vertical** scaling means that you scale by ***adding more powe***r (CPU, RAM) to an existing machine.

In a database world, **horizontal**-scaling is often based on ***partitioning*** of the data i.e. each node contains only part of the data , in **vertical**-scaling the data resides on a single node and scaling is done through ***multi-core i.e. spreading the load between the CPU and RAM resources*** of that machine.

With horizontal-scaling it is often easier to scale dynamically by adding more machines into the existing pool - Vertical-scaling is often limited to the capacity of a single machine, scaling beyond that capacity often involves downtime and comes with an upper limit.

Conceptually, the two models are almost identical as in both cases we break a sequential piece of logic into smaller pieces that can be executed in parallel. Practically, however, the two models are fairly different from an implementation and performance perspective. **The root of the difference is the existence (or lack) of a shared address space.** In a multi-threaded scenario you can assume the existence of a shared address space, and therefore data sharing and message passing can be done simply by passing a reference. In distributed computing, the lack of a shared address space makes this type of operation significantly more complex. Once you cross the boundaries of a single process you need to deal with partial failure and consistency. Also, the fact that you can’t simply pass an object by reference makes the process of sharing, passing or updating data significantly more costly (compared with in-process reference passing), as you have to deal with passing of copies of the data which involves additional network and serialization and de-serialization overhead.

http://ht.ly/cAhPe

ACID vs CAP

ACID:
- A **transaction** is a ***bundling*** of one or more operations on database state into a single sequence, A true transaction must adhere to the ACID properties
- ACID transactions offer guarantees that absolve the end user of much of the headache of ***concurrent access to mutable database state***
- ACID stands for:
  - **Atomic**: ***single action***, all are completed or none are
  - **Consistent**: follow the defined ***rules and restrictions*** of the database - constraints, triggers, cascades
  - **Isolated**: concurrency control, system state that would be obtained if transactions were executed ***serially***
  - **Durable**: ***persist*** and will not be undone

CAP:
- A tool to explain ***trade-offs in distributed systems***
- CAP stands for:
  - **Consistent**: All replicas of the same data will have the ***same value across*** a distributed system.
  - **Available**: All ***live nodes*** in a distributed system can process operations and ***respond to queries***.
  - **Partition Tolerant**: The system is designed to operate in the face of unplanned network connectivity loss between replicas.
- CAP isn't about what is possible, but rather, what isn't possible
- A more practical way to think about CAP: In the face of network partitions, you can't always have both perfect consistency and 100% availability. Plan accordingly.
- We can't ignore partitions. *If you don't have partitions, then you don't have a distributed system*

- ACID applies to transaction and databases, CAP applies to distributed systems which may involve application in addition to database
- **The main difference is in Consistency**:
  - **ACID** consistency is all about ***database rules enforcement***.
  - **CAP** consistency promises that ***every replica of the same logical value***, spread across nodes in a distributed system, has the same exact value at all times. Due to the speed of light, it may take some non-zero time to replicate values across a cluster.

A **BASE** system gives up on consistency so as to have greater Availability and Partition tolerance. A BASE can be defined as following:
Basically Available indicates that the system does guarantee availability.
Soft state indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
Eventual consistency indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

NoSQL Types and Selection Criteria

Types:
1. **Key-Value Store** – It has a Big Hash Table of keys & values {Riak, DynamoDB}
2. **Document-based Store**- It stores documents made up of tagged elements. {MongoDB, CouchDB}
3. **Column-based Store**- Each storage block contains data from only one column, {HBase, Cassandra}
4. **Graph-based**- A network database that uses edges and nodes to represent and store data. {Neo4J}

Selection Criteria

Storage Type:
- For instance, **get, put and delete** functions are best supported by ***Key Value*** systems.
- **Aggregation** becomes much easier while using ***Column oriented*** systems as against the conventional row oriented databases. They use tables but do not have joins.
- **Mapping** data becomes easy from object oriented software using a ***Document*** oriented NoSQL database such as XML or JSON as they use structure document formats.
- Tabular format is replaced and data is stored in ***graphical*** format by display **relationships**.

Concurrency Control:
- ***Locks*** prevent more than one active user to edit an entity such as a document, row or an object.
- ***MVCC (Multi-Version Concurrency Control)***, guarantee a **read consistent view** of the database, but result in conflicting versions of an entity if multiple users modify it at once. MVCC makes it possible for a transaction to seamlessly go through by maintaining many **different versions of the object**. That means transaction consistency is maintained even if that shows varying snapshots to different users at any given point in time. Any changes made to the database will be shown to others depending which snapshot are they referring to.
- ***None*** – Atomicity is missing in some systems thereby not providing the same view of the database to multiple users editing the database.
- ***ACID*** – For reliable database transactions, ACID or Atomicity, Consistency, Isolation, Durability is a safe bet. It allows for **pre-screening transactions** to avoid conflicts with no deadlocks.

Replication: Replication ensures that mirror copies are always in sync.
- ***Synchronous Mode*** – **expensive** approach as there is a dependency on the second server to respond, but it always **ensures consistency**. This ensures data is placed in multiple nodes at the same time.

- ***Asynchronous mode***- In this mode, one database gets updated without waiting for answer from the other database. Two databases could be not consistent in the range of few milliseconds. **cost-effective and 'Eventually Consistent.'**

NoSQL Types and Use cases

Key-value databases

- well-suited to applications:
 - frequent small reads and writes
 - simple data models
 - values stored may be simple scalar values, such as int or bool, but they may be structured data types too

- lacks in:
 - complex query capabilities
 - query the database by specific data value.
 - storing relationships between data values.
 - operate on multiple unique keys.
 - updating a part of the value frequently.

- Use cases:
 - Caching data from relational databases to improve performance - billing data
 - Tracking transient attributes in a Web application, such as a shopping cart
 - Storing configuration and user data information for mobile applications
 - Storing metadata of large objects, such as images and audio files- clips

Document datastores:

- well-suited to applications:
 - flexibility - store varying attributes along with large amounts of data
 - embedded documents - denormalizing - frequently queried data is stored together
 - improved query capabilities - with indexing and the ability to filter documents based on attributes

- Use cases:
 - Back-end support for websites with high volumes of reads and writes
 - Managing data types with variable attributes, such as products
 - Tracking variable types of metadata
 - Applications that use JSON data structures
 - Applications benefiting from denormalization by embedding structures within structures

Column Family datastores:
- well-suited for:
 - aggregated data
- column-wise access (instead of select * from table of 10 columns u always do select 1,2)
- data compression

- lacks in:
 - high consistency

- Use cases:
 - Security analytics using network traffic and log data mode
 - Big Science, such as bioinformatics using genetic and proteomic data
 - Stock market analysis using trade data
 - Web-scale applications such as search
 - Social network services

Graph datastores:
- well-suited for:
 - representations as networks of connected entities
 - if instances of entities have relations to other instances of entities

- Use cases:
 - Network and IT infrastructure management
 - Identity and access management
 - Business process management
 - Recommending products and services
 - Social networking

MongoDB

Data Modeling in MongoDB
- Model according to application's data access patterns
- Everything boils down to : What is the cardinality of the relationship?

Modeling One-to-Few:
- This is a good use case for embedding
  – you'd put the addresses in an array inside of your Person object
  - advantage : you don't have to perform a separate query to get the embedded details
  - disadvantage : you have no way of accessing the embedded details as stand-alone entities

One-to-Many:
- This is a good use case for referencing
  – you'd put the ObjectIDs of the Parts in an array in Product document
  - advantage : Each Part is a stand-alone document, so it's easy to search them and update them independently
  - disadvantage: having to perform a second query to get details about the Parts for a Product (application-level join)

One-to-Squillions:
- This is the classic use case for "parent-referencing" – you'd have a document for the host, and then store the ObjectID of the host in the documents for the log messages

// Choose one of the above and try to denormalize it
// Bi-directional referencing & Denormalizing to avoid joins at the cost of not having atomic updates

Two-Way Referencing:
- combine two techniques and include both styles of reference in your schema
- e.g Person <-> Task
- advantage : quick and easy to find the Task's owner and all tasks for a person
- disadvantage : to re assign the task to another person, you need to perform two updates

Denormalizing With "One-To-Many" Relationships:

Denormalizing from Many -> One :
  - eliminate application-level join in certain cases, at the price of some complexity with updates
  - For the parts example, you could denormalize the name of the part into the 'parts[]' array
    - saves you a lookup of the denormalized data at the cost of a more expensive update

- Denormalizing only makes sense when there’s a high ratio of reads to updates
- Note: if you denormalize a field, you lose the ability to perform atomic and isolated updates on that field

Denormalizing from One -> Many :
- product id and catalogue in part
- it’s significantly more important to consider the read-to-write ratio when denormalizing in this way

Denormalizing With “One-To-Squillions” Relationships:
- you can either put information about the “one” side (from the 'hosts’ document) into the “squillions” side (the log entries)
- if there’s only a limited amount of information you want to store at the “one” side, you can denormalize it ALL into the “squillions” side and get rid of the “one” collection altogether
- you can put summary information from the “squillions” side into the “one” side

DynamoDB:

Read/Write Performance and Scaling:
- Defined in terms of *Throughput Capacity*
- has to provisioned earlier to ensure consistent, low-latency performance
- One **read capacity** unit represents one strongly consistent read per second, or two eventually consistent reads per second, for an item up to **4 KB** in size
- One **write capacity** unit represents one write per second for an item up to **1 KB** in size

- If capacity limits exceed - requests throttle
  - The AWS SDKs have built-in support for retrying throttled requests
  - Auto scaling actively manages throughput capacity for tables and global secondary indexes

Replication:
- DynamoDB data is replicated across multiple nodes in different availability zones in a region
  - When a write data receives an HTTP 200 response (OK), all copies of the data are updated
  - The data is eventually consistent across all storage locations, usually within one second or less

Partitioning:
- Partition is an allocation of storage for a table, backed by solid-state drives (SSDs) and automatically replicated across multiple Availability Zones
- The data in a GSI is stored separately from the data in its base table
- **Details:**
  - A single partition can support a maximum of 3,000 read capacity units or 1,000 write capacity units
  - Initial number of partitions can be expressed as follows:
    (readCapacityUnits / 3,000) + (writeCapacityUnits / 1,000) = initialPartitions
  - A ***partition split*** can occur in response to:
    - Increased provisioned throughput settings
      - double the number of partitions
    - Increased storage requirements
      - split that partition only
  - DynamoDB would split the partition as follows:
    - Allocate two new partitions (P1 and P2)
    - Distribute the data from P evenly across P1 and P2.
    - Deallocate P from the table.

Burst Capacity:
- retains up to five minutes of unused read and write capacity

 - During an occasional burst of read or write activity, these extra capacity units can be consumed very quickly—even faster than the per-second provisioned throughput capacity

Data Distribution:
- Primary key = *Partition Key*
 - stores and retrieves each item based on its partition key value
- Composite Primary key = *Partition key + Sort/Range key*
 - stores all of the items with the same partition key value physically close together, ordered by sort key
 - there is no upper limit on the number of distinct sort key values per partition key value

Best Practices:
- Design For **Uniform Data Access** Across Items In Your Tables
 - (*if you anticipate scaling beyond a single partition) you will utilize your throughput more efficiently as the ratio of partition key values accessed to the total number of partition key values in a table grows
- **Randomizing the writes** Across Multiple Partition Key Values
- Use **One-to-Many Tables Instead Of Large Set Attributes**
 - advantages : can grow independently, easy read with specific filters and updates
 - disadvantages: may incur extra hop for reads
- Storing large attributes:
 - compress them or store them in s3 or break them up across Multiple Items

Dynamodb Internal Details

Concurrency vs Parallelism

Concurrency: Interruptibility, multitasking on single cpu
Parallelism: Independent Ability, multiprocessing on multicore machine

So what do I mean by above definitions? I will clarify with a real world analogy. Let's say you have to get done 2 very important tasks in one day

Get a passport
Get a presentation done
Now problem is task-1 requires you to goto an extremely bureaucratic government office that makes you wait for 4 hours in a line to get your passport. Whereas task-2 is required to be done for your office and it is a critical one. Both of these have to be finished on a specific day.

Case-1: Sequential Execution Ordinarily, you will drive to passport office for 2 hours, wait in the line for 4 hours, get the task done, drive back two hours, go home, stay away 5 more hours and get presentation done.

(You single handedly do start 2 tasks simultaneously pausing one wherever possible)
Case-2: ***Concurrent*** Execution: But you are smart. You plan ahead. What you do is, you carry a laptop with you, and while waiting in the line, you start working on your presentation. This way, once you get back at home, you just need to work one extra hour instead of 5 more hours. In this case, both the tasks are done by you, just in pieces. You interrupted the passport task while waiting in the line and worked on presentation. Whereas when your number was called, you interrupted presentation task and switched to passport task. The saving in time was essentially possible due to interruptibility of both the tasks. **Concurrency, IMO, should be taken as "isolation" in ACID properties of a database**.Two database transactions satisfy isolation requirement if you perform sub-transactions in each in any interleaved way and the final result is same as if the two tasks were done serially. Remember, that for both the passport and presentation tasks, you are the sole executioner.

(You appoint someone else to do some independent task)
Case-3: ***Parallel*** Execution Now since you are such a smart fella, obviously you are a higher up and you have got an assistant. Now before you leave to do passport task, you call him and tell him to prepare first draft of the presentation. You spend your entire day and finish passport task, come back and see your mails and you find the presentation draft. He has done a pretty solid job and with some edits in 2 more hours, you finalize it. Now since, your assistant is just as smart as you, he was able to work on it independently without a need to ask you for constant clarifications. Thus, the due to the independentability of the tasks, they were performed in the same time by two different executioners.

Concurrency in Java:

- A Java application runs by default in one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.
- Limits of concurrency gains
  - Within an application you work with several threads to achieve parallel processing or asynchronous behavior
  - the runtime is limited by parts of the task which can be performed in parallel
- ***Concurrency issues***
  - Threads have their own call stack, but can also access shared data
  - Therefore you have two basic problems, visibility and access problems
    - A visibility problem : if thread A reads shared data which is later changed by thread B and thread A is unaware of this change
    - An access problem : if several thread access and change the same shared data at the same time
    - Visibility and access problem can lead to
      - Liveness failure: The program does not react anymore due to problems in the concurrent access of data, e.g. deadlocks
      - Safety failure: The program creates incorrect data

- ***synchronized keyword***
  - provides locks to protect certain parts of the code to be executed by several threads
  - ensures
    - only a single thread can execute a block of code at the same time
    - each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock
  - Synchronization : necessary for mutually exclusive access to blocks and reliable communication between threads
  - synchronized keyword for the definition of a method
    - only one thread can enter
    - another threads which is calling this method would wait

- ***volatile keyword***
  - guaranteed that any thread that reads the field will see the most recently written value
  - will not perform any mutual exclusive lock on the variable

- ***Atomic operation***
  - performed as a single unit of work without the possibility of interference from other operations
  - reading or writing a variable is an atomic operation(unless the variable is of type long or double
  - The i++ (increment) operation it not an atomic operation
  - Since Java 1.5 the java language provides atomic variables, e.g. AtomicInteger or AtomicLong

- ***Immutability***
  - avoid problems with concurrency
  - To make a class immutable make
    - all its fields final
    - the class declared as final
    - the this reference is not allowed to escape during construction
    - Any fields which refer to mutable data objects are private
    - have no setter method
    - they are never directly returned of otherwise exposed to a caller
    - if they are changed internally in the class this change is not visible and has no effect outside of the class
  - For all mutable fields, e.g. Arrays, that are passed from the outside to the class - make a defensive-copy

- ***Defensive Copies***
  - You must protect your classes from calling code
  - To protect your class against that you should copy data you receive and only return copies of data to calling code
  - e.g. List passed from outside

- ***Threads in Java***
  - "Runnable" is the task to perform, "Thread" is the worker who is doing this task
  - A Thread executes an object of type java.lang.Runnable
  - Runnable is an interface with defines the run() method which is called by the Thread object
  - Using the Thread class directly has the following disadvantages
    - Creating a new thread causes some performance overhead
    - Too many threads can lead to reduced performance, context-switching
    - cannot easily control the number of threads, therefore you may run into out of memory errors

- ***Threads pools with the Executor Framework***
  - Thread pools manage a pool of worker threads
  - The thread pools contains a work queue which holds tasks waiting to get executed
  - thread pool can be described as a collection of Runnable objects
  - Instead of creating new threads when new tasks arrive, a thread pool keeps a number of idle threads that are ready for executing tasks as needed
  - After a thread completes execution of a task, it does not die, it remains idle waiting for work
  - The Java Concurrency API supports the following types of thread pools:
    - Cached thread pool: keeps a number of alive threads and creates new ones as needed.
    - Fixed thread pool: limits the maximum number of concurrent threads. Additional tasks are waiting in a queue.
    - Single-threaded pool: keeps only one thread executing one task at a time.

- Fork/Join pool: a special thread pool that uses the Fork/Join framework to take advantage of multiple processors to perform heavy work faster by breaking the work into smaller pieces recursively.
- e.g. web frameworks, database apps

- Implementation a thread pool = Executor
- The Executor framework provides example implementation of the java.util.concurrent.Executor interface, e.g. Executors.newFixedThreadPool(int n) which will create n worker threads
- ExecutorService adds life cycle methods to the Executor, which allows to shutdown the Executor and to wait for termination

- ***Runnable cannot return a result to the caller***

- ***Futures and Callables***
  - threads to return a computed result you can use java.util.concurrent.Callable which allows to return values after completion
  - uses generics to define the type of object which is returned
  - If you submit a Callable object to an Executor, the framework returns an object of type java.util.concurrent.Future
  - Future object can be used to check the status of a Callable and also be used to retrieve the result from the Callable
  - Future does not provide the option to register a callback method

- ***CompletableFuture***
  - allows to provide a callback interface which is called once a task is completed
  - support both blocking(wait) and nonblocking(callbacks) approaches, including regular callbacks

- ***Deadlock***
  - A concurrent application has the risk of a deadlock. A set of processes are deadlocked if all processes are waiting for an event which another process in the same set has to cause

Streams:

- A stream is **an abstraction, it’s not a data structure**
- The most important difference between a stream and a structure/collections is that a stream **doesn’t hold the data**
- A stream is an abstraction of a non-mutable collection of functions applied in some order to the data
- A stream represents **a pipeline through which the data** will flow and the functions to operate on the data

- The streams API gives us the power to **specify a sequence of operations** on the data in **individual steps**
- The code is so natural, we just follow the specification of what we have to do at every step
- The code is completely **unaware of the iteration logic** in the background
- they are **lazily evaluated**
- Some operations on the streams, particularly the functions that return an instance of the stream: filter, map, are called intermediate. This means that they won’t be evaluated when they are specified. Instead the computation will happen when the result of that operation is necessary.

- There are **pitfalls of running every stream operation in parallel**, because most streams implementations use the default **ForkJoinPool** to perform the operations in background. Thus, you can easily make the particular stream processing a bit faster, but instead sacrifice the **performance of the whole JVM** without even realizing it!

Collection Vs Stream:
- Collections are **in-memory data structures** which **hold elements** within it. Each element in the collection is **computed before** it actually becomes a part of that collection

Microservices:

How to refactor a Monolith to Microservice (the Soundcloud's way):

Monolith

- Pattern
  - **Adv**
    - Simple to develop - lot of tools, IDEs and Devs available
    - Simple to deploy - simple JAR/WAR in specific runtime
    - Simple to scale - running multiple copies of the application behind a load balancer
  - **Disadv**
    - Development
      - Difficult to understand and modify the large codebase
      - Overloaded IDE
      - Obstacle to scaling development - team split is difficult
      - Requires a long-term commitment to a technology stack
    - one misbehaving component can bring down the entire system
    - Overloaded web container - the larger the application the longer it takes to start up
    - Scaling the application can be difficult
      - a monolithic architecture is that it can only scale in one dimension, multiple servers. but the data remains shared and can become the bottleneck.
      - we cannot scale each component independently
    - Continuous deployment is difficult - In order to update one component you have to redeploy the entire application

Microservice

- Pattern
  - **Adv**
    - smaller services enables - easy testing, development, deployment and scaling
    - easy to understand, IDEs work faster, application boot time is small
    - Improved fault isolation
    - Eliminates any long-term commitment to a technology stack
  - **Disadv**
    - communication between services has to be handled carefully and may introduce some latency
    - data consistency
    - separate data stores for each service
    - 2 phase-commit/distributed transactions is not an option for many applications
    - An application must instead use the Saga pattern. A service publishes an event when its data changes. Other services consume that event and update their data
    - queries spanning multiple services
    - CQRS or API Composition

- operational complexity of deploying and managing a system (observability and traceability)
- Increased memory consumption - replaces N monolithic application instances with NxM services instances
- development
  - coordination between teams
  - one Must understand distributed systems paradigm

Components

- ***Communication style***
  - **Sync**
    - Remote Procedure Invocation - **REST**,gRPC
      - Simple and familiar
      - Client needs to discover locations of service instances
      - only supports request/reply and not other interaction patterns
      - Reduced availability - the client and service must be available for the duration of the interaction
  - **Async**
    - **Messaging** - using message brokers like Kafka or RabbitMQ or ActiveMQ
      - Loose coupling
      - Increased availability
      - Request/reply-style communication is more complex
      - Complexities of involving a broker

- ***Service Discovery and Load Balancing***
  - Why?
    - Each instance of a service exposes a remote API at a particular location (host and port)
    - The number of services instances and their locations changes dynamically
    - Virtual machines and containers are usually assigned dynamic IP addresses

  - **Service registry**
    - a database of services, their instances and their locations
    - healthcheck APIs to keep the data updated
    - e.g. Apache Zookeeper, Consul, Etcd
    - Disadv
      - yet another infrastructure component to manage
    - For example, Netflix Eureka service instances are typically deployed using elastic IP addresses. The available pool of Elastic IP addresses is configured using either a properties file or via DNS. When a Eureka instance starts up it consults the configuration to determine which available Elastic IP address to use. A Eureka client is also configured with the pool of Elastic IP addresses

- **Client-side discovery**
  - the client obtains the location of a service instance by querying a Service Registry, which knows the locations of all service instances
  - Disadv
    - need to implement client-side service discovery logic for each programming language/framework used by your application
    - couples the client to the Service Registry
  - For example, The service discovery is implemented using Netflix OSS components. It provides Eureka, which is a Service Registry, and Ribbon, which is an HTTP client that queries Eureka in order to route HTTP requests to an available service instance

- **Server-side discovery**
  - the client makes a request via a router (a.k.a load balancer) that runs at a well known location
  - An AWS Elastic Load Balancer (ELB) is an example of a server-side discovery router
  - An ELB also functions as a Service Registry
  - Disadv
    - yet another infrastructure component to manage
    - The router must support the necessary communication protocols

- ***Resilience and Fault tolerance***
- **Circuit breaker**
  - Why?
    - to prevent a network or service failure from **cascading** to other services
  - Solution
    - service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker
    - number of consecutive failures crosses a **threshold**, the circuit breaker trips
    - for the duration of the timeout period all attempts to invoke the remote service will fail immediately
    - after that time out circuit breaker tries to recover using small number requests and diff other strategies
  - e.g. Netflix **Hystrix**

- ***External API***
- **API Gateway Or BFF**
  - Why?
    - services typically provide fine-grained APIs, which means that clients need to interact with multiple services
    - Different clients need different data
    - Network performance is different for different types of clients
      - The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few

- number of service instances and their locations (host+port) changes dynamically
    - Partitioning into services can change over time and should be hidden from clients
    - Services might use a diverse set of protocols, some of which might not be client friendly
  - Solution
    - **API gateway**
      - **single entry point for all clients**
      - handles requests in one of two ways
        - simply proxied/routed to the appropriate service
        - fan out to multiple services
        - might also implement security e.g. authorization
    - **BFF**
      - **client-specific adapter code** that provides each client with an API that's best suited to its requirements
      - a separate API gateway for each kind of client
  - Disadv
    - increased response time due to the additional network hop through the API gateway
    - yet another moving part that must be developed, deployed and managed
  - How implement the API gateway? - An event-driven/reactive approach is best if it must scale to scale to handle high loads

- ***Data management***
  - Event sourcing
  - Transactions - SAGA

- ***Observability***
  - **Log aggregation**
    - Use a **centralized** logging service that aggregates logs from each service instance
    - AWS cloudwatch

  - **Distributed tracing**
    - Why?
      - to understand the behavior of an application and troubleshoot problems
      - External monitoring only tells you the overall response time and number of invocations - no insight
      - Log entries for a request are scattered across numerous logs
    - Solution
      - **Instrument** services with to assign each external request a unique external request id
      - **Pass the external request id** to all services that are involved in handling the request
      - Include the external request id in all log messages
      - trace using centralised log aggregation service

- e.g. Zipkin

- Exception tracking
- Health check API
- Application metrics
- Audit logging

- Security
  - Access Token

> **Detecting failures**

- Install a failure detector at load balancer or at Service discovery layer which use health endpoint(application failure) and other system data (system failure)
- Use a gossip protocol with reachability data - Serf by Hashicorp

> **Handling failures:**

Health-check and Load Balancing
Self-healing - replication, checkpointing, external systems to restart the service/node
Failover Caching
Retry Logic
Rate Limiters and Load Shedders

- **More ways in different words:**

Circuit Breaker
Timeout

Task Queue Vs. Message Queue

- RabbitMQ is a "MQ". It receives messages and delivers messages.

- Celery is a Task Queue. It receives tasks with their related data, runs them and delivers the results.

- Let's talk about RabbitMQ. What would we usually do? Our Django/Flask app would send a message to a queue. We will have some workers running which will be waiting for new messages in certain queues. When a new message arrives, it starts working and processes the tasks.

- Celery manages this entire process beautifully. We no longer need to learn or worry about the details of AMQP or RabbitMQ. We can use Redis or even a database (MySQL for example) as a message broker. Celery allows us to define "Tasks" with our worker codes. When we need to do something in the background (or even foreground), we can just call this task (for instant execution) or schedule this task for delayed processing. Celery would handle the message passing and running the tasks. It would launch workers which would know how to run your defined tasks and store the results. So you can later query the task result or even task progress when needed.

- You can use Celery as an alternative for cron job too (though I don't really like it)!

REST

REST (Representational State Transfer)
- is a **style of architecture** based on a set of principles that describe how networked resources are defined and addressed
- is a **set of guidelines as opposed to a set of standards**

An application or architecture considered RESTful or REST-style is characterized by:

- State and functionality are divided into distributed **resources**
- Every resource is uniquely **addressable** using a uniform and minimal set of commands (typically using HTTP commands of GET, POST, PUT, or DELETE over the Internet)
- The protocol is client/server, **stateless**, layered, and supports caching

The key principles of REST involve separating your API into logical resources. These resources are manipulated using HTTP requests where the method (GET, POST, PUT, PATCH, DELETE) has specific meaning.

Terminologies
- ***Resource***
  - is an object or representation of something, which has some associated data with it and there can be set of methods to operate on it
  - key here is to not leak irrelevant implementation details out to your resource naming
  - **why plural?**
    - again just a guideline
    - Representing endpoints as **paths on a file system** is the most expressive way of doing it. And within that context, linguistic rules can only get you as far as the following:

`A **directory** can contain multiple files and/or sub-directories, and therefore its name should be in plural form`
Although, on the other hand - it's your directory, you can name it "a-resource-or-multiple-resources" if that's what you want

- ***URL*** (Uniform Resource Locator)
  - is a path through which a resource can be located
  - The URL is a sentence, where resources are nouns and HTTP methods are verbs

- HTTP response status codes - categories

- Think about these while designing -> Searching, sorting, filtering, pagination, versioning and Caching

- ***API design gotchas***
  - What about **actions that don't fit into the world of CRUD** operations?
    1. **Restructure** the action to appear like a **field of a resource**. This works if the action doesn't take parameters. For example an activate action could be mapped to a boolean activated field and updated via a PATCH to the resource.
    2. Treat it like a **sub-resource** with RESTful principles. For example, GitHub's API lets you star a gist with PUT /gists/:id/star and unstar with DELETE /gists/:id/star
    3. Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource’s endpoint. In this case, /search would make the most sense even though it isn't a resource. This is OK - **just do what's right from the perspective of the API consumer** and make sure it's documented clearly to avoid confusion.
  - How to choose **headers vs query params**?
    - query params - should be used for the **purpose of filtering or action**(sort, embed) type of scenarios
    - header - info that **needs to be passed with almost every similar API call** or some chit-chat info

Advantages
- Easy to integrate
- use of ubiquitous standards - HTTP, Json
- Scalability – stateless communication, replicated repository make for a good scalability potential

Disadvantages
- limited number of verbs - example archives can’t be treated as true DELETE action
- stateless requires lots of piggybacking in every request
- nested resource fetch may require extra hops

https://hackernoon.com/restful-api-designing-guidelines-the-best-practices-60e1d954e7c9
https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

GraphQL vs REST

GraphQL
- is a **declarative data fetching specification** and a query language
- a common interface between the client and the server for data fetching and manipulations

Differences Between REST and GraphQL
- ***Data fetching***
 - *REST* may involve **multiple requests** for fetching nested resources
  e.g. Author with all the posts and comments on them
  - /authors/:id -> /authors/:id/posts -> /authors/:id/posts/:id/comments
 - *GraphQL* we only need to make **a request to one endpoint**, say /graphql with the following query:

```
  {
    author {
      name
      posts {
        title
        comments {
          comment
        }
      }
    }
  }
```

- ***Over/Under Fetching***
 - let's say you want only part of data
  - *REST* has a **fixed data structure** which it is meant to return whenever it is hit
   - client takes whatever it wants and ignores the rest
  - with *GraphQL* we **only fetch what we need** from the server by constructing our query to only include what we need

- ***Error Handling***
 - REST - has specific **HTTP error codes**
 - GraphQL - will **return 200** but errors will have full description

- ***Caching***
 - REST - can be easily built with **HTTP caching**
 - GraphQL - has to implemented **manually**

- ***Versioning***
 - REST - **good to have** versioning
 - GraphQL - there is **no need**, easily add new fields and types to our GraphQL API without impacting existing queries

REVERSE PROXY VS. LOAD BALANCER

The basic **definitions** are simple:

A ***reverse proxy*** accepts a request from a client, forwards it to **a server** that can fulfill it, and returns the server's response to the client.
A ***load balancer*** distributes incoming client requests among a **group of servers**, in each case returning the response from the selected server to the appropriate client.

Load Balancing
- commonly deployed when a site needs **multiple servers**
- eliminates a single point of failure
- **distribute the workload** in a way that makes the best use of each server's capacity, results in the fastest possible response to the client
- **enhance the user experience** by detecting unhealthy servers and redirecting request to healthy ones
- session persistence, which means sending all requests from a particular client to the same server

Reverse Proxy
- makes sense to deploy a reverse proxy even with just one web server or application server
- website's "**public face**"
- Increased **security** - no one knows backend servers, softwares for protecting from DDOS
- Increased **scalability and flexibilit**y - can change backend architecture
- **web acceleration** - do common tasks(compression/de, encryption/un, caching) at this layer

Clean Code

What is Clean Code?
Code is clean if it can be understood easily – by everyone on the team.
With understandability comes readability, changeability, extensibility and maintainability.

Why?
- responsiveness to change increases
- bugs are not hidden

Principles?
- Loose Coupling - components should not depend on each other too much
- High cohesion - all dependent things should stay close
- Change is Local - there are boundaries in the design which changes should not cross
- Easy to Remove Components - should be able to delete or extract components easily

Code Smells?
- Bloaters - large everything, primitive obsession
- Change Preventers - requiring many changes or same change in many places
- Dispensables - unused, duplicate code
- Couplers
- Object-Orientation Abusers

SOLID
- Single Responsibility Principle
  - Class should have only one responsibility - highly cohesive and implement strongly related logic
- Open Closed Principle - open for extension but closed for modification
- Liskov Substitution Principle
  - derived types must be completely substitutable for their base types
  - should not change the behavior of the base class
- Interface Segregation Principle - Client should not depend on interface/methods which it is not using
- Dependency Inversion Principle- Depend on abstractions, not on concretions

General
- Follow Standard Conventions
- Keep it Simple, Stupid (KISS) Simpler is always better - Reduce complexity as much as possible
- Boy Scout Rule Leave the campground cleaner than you found it
- Root Cause Analysis - always look for the root cause of a problem. Otherwise, it will get you again and again

From Legacy Code to Clean Code

- Always have a Running System - small steps, change from a running state to a running state.
- Identify Features
- Introduce Boundary Interfaces for Testability - Refactor the boundaries of your system to interfaces so that you can simulate the environment
- Write Feature Acceptance Tests
- Drill down components(likelihood and risk of change) - repeat above steps at this level
- Decide for Each Component: Refactor, Reengineer, Keep

Continuous Integration

- Pre-Commit Check - locally
- Post-Commit Check - on ci server
- Communicate Failed Integration to Whole Team
- Build Staging
- Automatically Build an Installer(Artifact) for Test System
- Continuous Deployment - test environment on every commit or manual request. Deployment to prod env is automated

Continuous Delivery

- From the customer's point of view, the essential benefit of the agile approach is that it allows software development to be a transparent process.
- Software grows in a visible way, so that businesses can learn from previous iterations when considering future work.
- In order to make this work fully you need a deployment pipeline that ensures that software is built in small production-ready increments.
- many production deployments every day. This allows a whole new relationship between the developers of software and their users and customers.

Consul

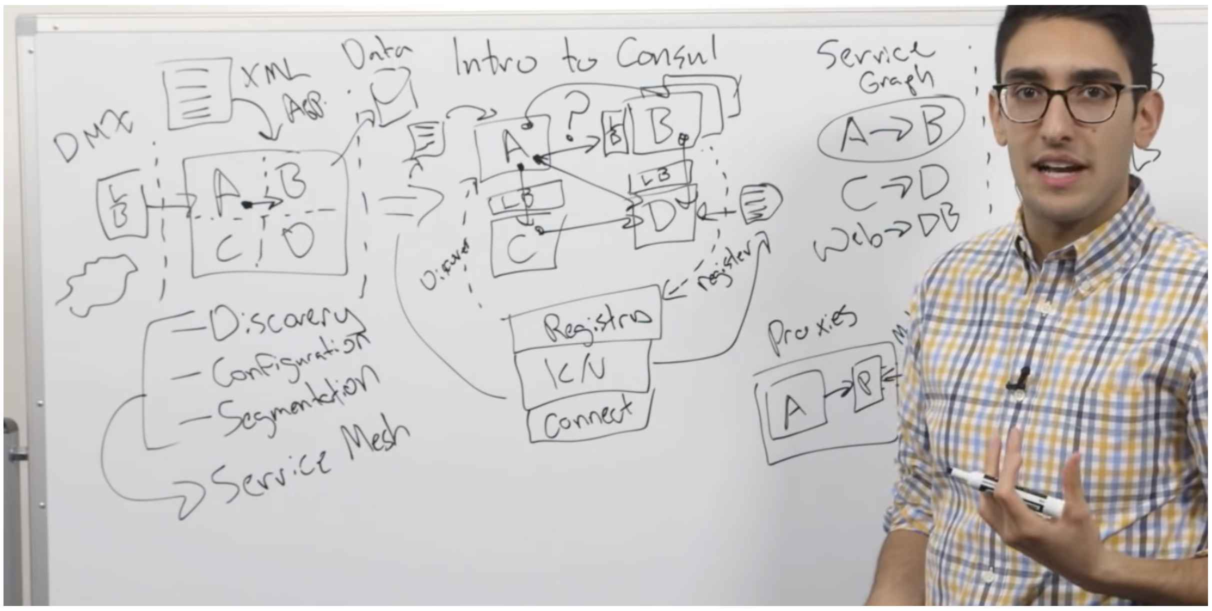
From their website - "Consul is a distributed service mesh to connect, secure, and configure services across any runtime platform and public or private cloud"

Provides ***Service Mesh*** capabilities which has basically 3 pillars,

Discovery
  - Common solution is to use load balancers in front of services and individual services talk to load balancers via hardcoded IP address (which allows services to scale behind LB but LBs add cost, increase latency, introduce single points of failure, and must be updated as services scale up/down)
  - Consul does this via central ***Service Registry*** which keeps track of real-time list of services, their location, and their health
  - When Services bootstrap they register with Consul and when they want to communicate with each other they contact Consul to get the address

Configuration
  - In monolith world there was single configuration(maintainance_mode=true/false) file shared with all
  - Now Consul provides ***global KV store*** which can be shared between all services

Segmentation
  - In monolith world we had 3 layer network segmentation - Web -> Application -> Database
  - Now in microservice world there are multiple segments and managing them is difficult due to communication between services
    - ***Service graph*** - keeps track of which service can talk to which other service
    - ***Proxies with TLS*** - ensures identity of services and keeps communication encrypted



External Service Communication

- Set timeout at the lowest layer
    - E.g. If you are using Hystrix with Retrofit , Hystrix will use threads to run its commands on which internally use some http client. So if you set timeout at thread/hystrix level, it will only interrupt the thread but the underlying http connection will be still open.
- Why to use queues with hystrix
    - There is a threshold on number of threads you can use for hystrix command. Generally you will use a special thread pool.
    - If all of these threads are occupied/waiting new requests will be rejected, to avoid this hystrix provides concept of queue so store these incoming request giving the thread/current executing request to complete and in turn give the corresponding external service time to heal if it’s just a temporary hiccup

Postgres

```
docker exec -it 6502fdd88d33 bash
psql -h localhost -U postgres
create database test_001;
\c test_001
CREATE TABLE users (id int, username text);
\q
psql -d test_001 -U postgres
```

How to debug slow db performance

1. Information Gathering
- specific queries, applications connected, timeframes
- check if something else was running at that time
  - regular job runs
  - heavy vacuum runs
  - system peak load

2. Check slow queries
- check for metrics if you have any
- check pg_stat_activity
  - check state - is waiting/active
  - wait_event_type - info about waiting - explore pg_locks

3. Taking help from query planner
- check using `explain`(only query plan)/ `explain analyze`(plan plus run) command
  - check for sequential scans (need indexes), nested loops, expensive sorting

4. Check the health of the tables
- check if tables are bloated (if you have deletes and updates in your queries)
  - run `vacuum` (w/o exclusive lock - returns space to usage in same table)
  - or `vacuum full` (with exclusive lock - returns space to OS)
  - or `vacuum analyse` (ANALYZE collects statistics to be used by query planner)

5. Check the health of the host system/ provider
- (if aws) check if you have used all the required iops
- check system load if running on-premise
  - free/swap memory
  - disk access bottlenecks

Java app performance

1. Identify goals
- avg response time
- avg concurrent users
- requests per second during peak load

2. Identify bottlenecks
- load test - Gatling - check cpu, memory, heap sizes
- app perf management - Retrace
- identify layer at which it is slowing down
  - dependency slow? - db/cache, external services

3. Code level improvements
- string vs string builder, regex, recursion
- threads - pools - fork-join pool

4. JVM tuning
- heap size
  - understand footprint - threads, classes, in-memory caching
  - start big and lower it down using monitoring over the time
- gc
  - make sure that u r not using stop-the-world gc
  - ****

5. JDBC performance (optional)
- use connection pooling
  - HikariCP JDBC – a very lightweight (~130Kb), lightning fast framework
- batching and caching
  - reduce network ops
  - cache frequently used prepared statements