Web Scalability for Startup Engineers Notes

1: Core Concepts

- Avoid full re-writes, they always cost and take more than expected and you end up with similar issues.
- Single-server configuration means having everything running off one machine, good option for small applications and can be scaled vertically.
- Vertical scaling gets more expensive in time and has a fixed ceiling.
- Isolation of services means running different components off different servers like DNS, cache, data storage, static assets, etc., which can then be vertically scaled independently.
- CDN
    - A hosted service that takes care of global distribution of static files like images, JavaScript, CSS, and videos. (Acts as an HTTP proxy.)
    - CDNs bring things like geoDNS for your static assets so requests will be routed to the geographically closest servers.
- High-level overview of the data center infrastructure (P23)
    - The Front Line: geoDNS, load balancer, front cache
    - Web Application Layer
        - Render user interface, should be completely stateless
        - Lightweight web framework with a minimal amount of business logic
    - Web Services Layer
        - Usually use REST/SOAP between front-end and web services
        - Also should be stateless
        - Additional components: object caches, message queues
    - Data Persistence Layer
        - data store, search
- Overview of the Application Architecture
    - Front End
        - Single responsibility: becoming the user interface
        - The layer translating between the public interface and internal service calls
        - Should not be aware of any databases or 3rd party services
    - Web Services
        - Service-oriented architecture
            - architecture centred around loosely coupled and autonomous services that solve specific business needs.
        - Layered architecture
            - architecture that divide functionality into layers where the lower layers provide an API for and don't know about the upper ones.
        - Hexagonal architecture
            - architecture that assumes the business logic is in the center and all interactions with other components, like data stores or users, are equal, everything outside the business logic requires a strict contract.
        - Event-driven architecture
            - different way of thinking about actions, is about reacting to events that already happened while traditional architectures are about responding to requests.

2: Principles of Good Software Design

- Simplicity.
    - Hide complexity behind abstractions.
    - Avoid overengineering.
    - Test Driven Development
    - SOLID principles
    - Don’t reinvent the wheel, follow iniefficient processes
- Promote Loose Coupling
    - Hiding details (private, protected) is a great way to reduce coupling and promote simplicity.
    - Drawing diagrams helps spot coupling.
- Draw Diagrams
    - A must-have skill for every architect and technical leader (switch from code first to DESIGN FIRST!)
        - Use Case Diagrams: actors, actions and high-level structure of how different operations relate to each other.
        - Class Diagrams: Visualize a module's structure with its classes, interfaces, and their interdependencies.
        - Module Diagrams
- Designing for Scale
    - Adding clones
        - good for stateless services, difficult to scale stateful servers
    - Functional Partitioning
        - Limited functional partitions that you can come up with, you can't keep rewriting your application and keep dividing it into smaller and smaller Web services.
    - Data Partitioning
- Design for self healing
    - removing single points of failure and graceful failover

3: Building the Front-End Layer

- Every user interaction, every connection, and every response has to go through the front-end layer in one form or another. This in turn causes the front-end layer to have the highest throughput and concurrency rate demands, making its scalability critical.
- You can have your frontend be a traditional multipage web application, SPA, or a hybrid.
- Managing State
    - Stateful vs. stateless
        - Web servers can be stateless (handle requests from every user indistinctly) or stateful (handle every request of some users).
        - state is any information that would have to be synchronized between servers to make them identical.
            - Example: Order drinks in a pub: cash (stateless), tab (stateful)
        - **Prefer stateless servers where session is not kept within the servers but pushed to another layer like a shared session storage which makes for better horizontal**

scalability since you can add/remove clones without affecting users bound sessions.

- HTTP is stateless but cookies makes it stateful.
  - On the first HTTP request the server can send a Set-Cookie:SID=XYZ... response header, after that, every consecutive request needs to contain the cookie Cookie:SID=XYZ...

- 1. Managing HTTP Sessions
  - The key thing to observe here is that any data you put into the session should be stored outside of the web server itself to be available from any web server.
  - Where can we store sessions?
    - a. Store session state in cookies
      - Session data can be stored in cookies: which means every request (css, images, etc) contains the full body of session data, when state changes, the server needs to re-send a Set-Cookie:SDATA=ABCDE... which increases request payload so only use if session state is small. As an advantage you don't have to store any session data in the server.
      - Pros
        - Do not have to store the session state anywhere in your data center.
      - Cons
        - Session storage becomes expensive.
    - b. Store session data in a dedicated data store.
      - Keep session state in shared data store: like Memcached, Redis, Cassandra; then you only append the session id to every request while the full session data is kept in the shared store which can be distributed and partitioned by session id.
      - Requires very low latency on get-by-key and put-by-key operations.
    - c. Use a load balancer that supports sticky sessions.
      - Or use a load balancer that supports sticky sessions: this is not flexible as it makes the load balancer know about every user, scaling is hard since you can't restart or decommission web servers without breaking users sessions.
      - Breaks the fundamental principle of statelessness.
- 2. Managing Files
  - Stick with a third-party provider like S3 first.
  - Opt for data store as a cheap alternative.
- 3. Managing Other Types of State
  - ○ Components of the Scalable Front End
    - Components of scalable frontends: DNS, CDN, Load Balancer, FE web server cluster.
    - a. DNS
      - DNS is the first component hit by users, use a third-party hosted service in almost all cases. There are geoDNS and latency based DNS services like Amazon Route 53 which can be better than the former cause they take into consideration realtime network congestion and outages in order to route traffic.
    - b. Load Balancers

        - Use load balancers as the entry point to your data center, allowing you to scale more easily and make changes to your infrastructure without negatively affecting your customers.
        - Load Balancers help with horizontal scaling since users don't hit web servers directly.
        - Can provide SSL offloading (or SSL termination) where the LB encrypts nd decrypts HTTPS connections and then your web servers talk HTTP among them internally.
        - Apart from an external LB routing traffic to FE web servers, you can have an internal LB to distribute from FE to web services instances and gain all the benefits of an LB internally.
        - Some LB route the TCP connections themselves allowing the use of other protocols apart from HTTP.
        - Benefits:
            - Hidden server maintenance
            - Seamlessly increase capacity
            - Efficient failure management
            - Automated scaling
            - Effective resource management / SSL offloading
            - Options
                - Load Balancer as a Hosted Service, e.g. ELB
                - Self-Managed Software-Based Load Balancer, e.g. Nginx / HAProxy
                - Hardware Load Balancer
    - c. Web Servers
    - d. Caching
        - Integrate a CDN
        - Deploy your own reverse proxy servers
        - Store data directly in the browser
        - Cache fragments of your responses in an object cache
    - e. Auto-Scaling

4: Web Services

- Designing Web Services (P124-131)
    - Web Services as an Alternative Presentation Layer
        - Monolithic Approach
            - Where you can have a single mvc web app containing all the web application controllers, mobile services controllers, third-party integration service controllers and shared business logic.
            - Here every time you need to integrate a new partner, you'd need to build that into the same mvc app as a set of controllers and views since there's not concept of separate web services layer.
        - API-First Approach
            - all clients talk to the web application using the same API which is a set of shared web services containing all the business logic.
        - Pragmatic Approach
- Types of Web Services

    - Function-Centric Services (SOAP)
        - concept of calling functions or objects in remote machines without the need to know how they're implemented
        - SOAP dominates this space, it uses WSDL files for describing methods and endpoints available and provide service discovery and an XSD files for describing data structure.
        - Your client code doesn't need to know that is calling a remote web service, it only needs to know the objects generated on web services contract (WSDL + XSD files).
        - Having a strict contract and the ability to discover functions and data types, provides a lot of value.
    - Resource-Centric Services (REST)
        - treat every resource as a type of object, you can model them as you like but the operations you can perform on them are standard (POST, GET, PUT, DELETE),
            - different from SOAP where you have arbitrary functions which take and produce arbitrary values.
        - REST is predictable, you know the operations will be always the same, while on SOAP each service had its conventions, standards and ws-* specifications.
        - Uses JSON rather than XML which is lighter and easier to read.
        - REST frameworks or containers is pretty much just a simple HTTP server that maps URLs to your code.
        - It doesn't provide discoverability and auto-generation of client code that SOAP has with WSDL and XSD but by being more flexible it allows for nonbreaking changes to be released in the server without the need to redeploy client code.
- Scaling REST Web Services
    - 1. Keeping Service Machines Stateless
        - No state held between HTTP requests other than auto-expiring caches.
    - 2. Caching Service Responses
        - The HTTP protocol requires all GET method calls to be read-only. A GET request to any resource should not cause any state changes or data updates.
        - Another important aspect to consider when designing a REST API is which resources require authentication and which do not.
    - 3. Functional Partitioning
        - The main challenge that may be an outcome of performing functional partitioning too early or of creating too man partitions is when new use cases arise that require a combination of data and features present in multiple web services.

5: Data Layer

- Scaling with MySQL
    - 1. Replication
        - Allows you to synchronize the state of master and slave
        - Can connect to a slave to read data from it
        - Can modify data only through the master server
        - One of the main reasons why people use replication in MySQL is to increase availability by reducing the time needed to replace the broken database.
        - Replication is ONLY applicable to scaling READS. It is NOT the way to scale WRITES
            - All of your writes will need to go through a single machine (or through each machine in case of multimaster deployments)

                - On MySQL you can have a master and slave topology, there's a limit on the slave count but you can then have multilayer slaves to increase the limit.
                - All writes go to master while replicas are read-only.
                - You can have a master-master topology for a faster/easier failover process but still not automatic in MySQL. In this topology, you write to either one and the other replicates from its binlog.
                - The more masters you have, the longer the replication lag, hence worst write times.
            - ○ Replication is not a way to scale the overall dataset size
                - All of the data must be present on each of the machines
            - ○ Slaves can return stale data
    - 2. Data Partitioning (Sharding)
        - ○ One of the most significant limitations that come with application-level sharding is that you cannot execute queries spanning multiple shards.
        - ○ ACID (Atomicity, Consistency, Isolation, Durability)
            - Another side effect of sharding is that you lose the ACID properties of your database as a whole.
        - When sharding, you want to keep together sets of data that will be accessed together and spread the load evenly among servers.
        - You can apply sharding to object caches, message queues, file systems, etc.
            - Cross-shard queries are a big challenge and should prob be avoided.
            - Cross-shard transactions are not ACID.
        - Try to avoid distributed transactions.
- Scaling with NoSQL
    - NoSQL databases make compromises in order to support their priority features.
    - CAP theorem: it is impossible to build a distributed system that would simultaneously guarantee consistency, availability and partition tolerance.
        - Consistency: all of the nodes see the same data at the same time
        - Availability: any available nodes can serve client requests even when other nodes fail
        - Partition tolerance: the system can operate even in the face of network failures where communication between nodes is impossible
    - ○ Eventual consistency
        - different nodes may have different versions of the data, but where state changes eventually propagate to all of the servers
        - These systems favor availability, they give no guarantees that the data you're getting is the freshest.
        - Amazon Dynamo (designed to support the checkout process) works this way, it saves all conflicts and sends them to the client where the reconciliation happens which results on both shopping cart versions being merged. They prefer showing previously removed item in a shopping cart than loosing data.
        - Cassandra topology: all nodes are the same, they all accept reads and writes, when designing your cluster you decide how many nodes you want the data to be replicated to which happens automatically, all nodes know which has what data.
            - It's a mix of Google's BigTable and Amazon's Dynamo, it has huge tables that most of the times are not related, rows can be partitioned by the row key among nodes and you can add columns on the fly, not all rows need to have all the columns.

6: Caching

- Cache hit ratio is the most important metric of caching which is affected by 3 factors.
    - data set size - the more unique your cache keys, the less chance of reusing them
    - space - how big are the data objects.
    - longevity - TTL (time to live).
- HTTP caches are read-through caches.
    - HTTP Caching Headers: Cache-Control, Expires, Vary
        - For these you can use request/response headers or HTML metatags, try to avoid the latter to prevent confusion.
        - Favor "Expires: A-DATE" over "Cache-Control: max-age=TTL", the latter is inconsistent and less backwards compatible.
    - 4 Types of HTTP Cache Technologies
        - Browser cache
            - exists in most browsers.
        - Caching Proxies
            - usually a local server in your office or ISP to reduce internet traffic, less used nowadays cause of cheaper internet prices.
            - HTTPS prevents caching on intermediary proxies.
        - Reverse Proxy
            - reduces the load on your web servers and can be installed in the same machine as the load balancer like Nginx or Varnish.
        - CDNs
            - Can also act as caches
    - Most caching systems use Least Recently Used (LRU) algorithm to reclaim memory space.
- Object Caches are cache-aside caches.
    - Common Types of Object Caches
        - Client-Side Caches
            - Client side like web storage specification
        - Caches co-located with code
        - Distributed Object Caches (Redis / Memcached)
            - ease the load on data stores, you app will check them before making a request to the db, you can decide if updating the cache on read or write.
            - scale object caches with consistent hashing
- Caching Rules of Thumb
    - Cache High Up the Call Stack
    - Reuse Cache Among Users
    - Where to Start Caching?
        - Start caching on aggregate time spent
    - Caching invalidation is hard:
        - if you were caching each search result on an e-commerce site, then updated one product, there's no easy way of invalidating the cache without running each of the search queries and see if the product is included.
        - The best approach often is to set a short TTL.

7: Asynchronous Processing

- Message Queues
    - Message Producers
        - Message producers are clients issuing requests (messages).
    - Message Broker
        - Message Broker | Event Service Bus (ESB) | Message-Oriented Middleware (MOM): app that handles message queuing, routing and delivery, often optimised for concurrency and throughput.
    - Message Consumers
        - Message Consumers are your servers which process messages.
        - Two most common ways "cron like", "daemon like"
            - cron-like: connects periodically to the queue and checks its status (pull model)
            - daemon-like: runs in an infinite loop, permanent connection to the message broker (push model)
    - Message routing methods
        - a. Direct Worker Queue: Each message arriving to the queue is routed to only one customer.
        - b. Publish / Subscribe
            - Producers publish messages to a topic, not a queue
            - Consumers declare which topics they are interested in
        - c. Custom Routing Rules
    - Messaging Protocols
        - a. AMQP: Advanced Message Queuing Protocol
            - AMQP is recommended, standardised (equally implemented among providers) well-defined contract for publishing, consuming and transferring messages, provides delivery guarantees, transactions, etc.
        - b. STOMP: Streaming Text-Oriented Messaging Protocol
            - STOMP, simple and text based like HTTP, adds little overhead but advanced features require the use of extensions via custom headers which are non-standard like prefetch-count.
        - c. JMS: Java Message Service
            - only for Java/JVM systems
    - Messaging Infrastructure
- Benefits of Message Queues
    - 1. Enabling asynchronous processing
        - Message queues only add values if your application is NOT built in an async way to begin with (e.g. message queues won't benefit you that much if you develop in Node.js)
    - 2. Easier Scalability
    - 3. Evening Out Traffic Spikes
        - if you get more requests, the queue just gets longer and messages take longer to get picked up but consumers will still process the same amount of messages and eventually even out the queue.
    - 4. Isolating Failures and Self-Healing
    - 5. Decoupling
        - Don't couple producers with consumers
- Message Queue-Related Challenges
    - a. No Message Ordering
        - But, having group ISs can help with get message order accurately/closely

        - some providers will guarantee ordered delivery of messages of same group
        - can watch out for consumer idle time.
    - b. Message Requeueing
        - strive for idempotent consumers
    - c. Race Conditions Become More Likely
    - d. Risk of Increased Complexity
- Message Queue-Related Anti-Patterns
    - a. Treating the MQ as a TCP Socket (Instead we should avoid return channels)
    - b. Treating MQ as a Database
    - c. Coupling Message Producers with Consumers
        - Don't do this
    - d. Lack of Poison Message Handling
- Event sourcing
    - technique where every change to the application state is persisted in the form of an event, usually tracked in a log file, so at any point in time you can replay back the events and get to an specific state, MySQL replication with binary log files is an example.

8: Searching for Data

- High cardinality fields (many unique values) are good index candidates cause they narrow down searches.
- Distribution factor also affects the index.
- You can have compound indexes to combine a high cardinality field and low distributed one.
- Inverted index: allows search of phrases or words in full-text-search.
    - These break down words into tokens and store next to them the document ids that contain them.
    - When making a search you get the posting lists of each word, merge them and then find the intersections.

9: Other Dimensions of Scalability

- Automate testing, builds, releases, monitoring.
    - Build Deployment - unit tests for >85% code coverage, E2E test cases for all critical paths, feature toggles, AB tests
    - Monitoring and Alerting - Reducing mean time to recovery (MTTR)MTTR = time to discover + time to respond + time to investigate + time to fix
    - Log aggregation
- Scale yourself
    - Overtime is not scaling
    - Recovering from burnout can take months
    - Use 3rd-party services or commercial tools
- Project management levers: Scope, Time, Cost
    - When modifying one, the others should recalibrate.
    - Influencing scope can be the easiest way to balance workload.

- "Without data you're just another person with an opinion".
- Stay pragmatic using the 80/20 heuristic rule.