Kenny R N Wilson

System Design

Introduction

THIS DOCUMENT COVERS

- Introduction

Cheat Sheets

Measuring Time

❶ **Millisecond**	One thousandth of a second	$10^{-3} = {}^{1}/_{1,000} = 0.001$
❷ **Microsecond**	One millionth of a second	$10^{-6} = {}^{1}/_{1,000,000} = 0.000001$
❸ **Nanosecond**	On billionth of a second	$10^{-9} = {}^{1}/_{1,000,000,000} = 0.000000001$

Measuring Space

❶ **1K**	1024 Bytes	$\sim 1\ thousand\ bytes$
❷ **1MB**	1,048,576	$\sim 1\ million\ bytes$
❸ **1GB**	1,077,741,824	$\sim 1\ billion\ bytes$
❹ **1TB**	1,099,511,627,776	$\sim 1\ trillion\ bytes$

Non-Functional Requirements / Desirable Properties

Reliability

A reliable system is one in which faults do not lead to failure. We say the system is fault-tolerant or resilient. Faults can be classified as hardware faults, human errors, and software errors.

Hardware Faults

Hard disk crashes, memory faults and power outages are all good examples of hardware faults. We can arrange our disks in raid configuration, duplicate power supplies etc to minimise the chance of a single machine going down however modern systems are moving more and more to architectures that are tolerant of whole machines going down.

To calculate availability, see the following

https://static.usenix.org/event/osdi10/tech/full_papers/Ford.pdf

Software Faults

The only way to deal with software faults is to perform careful testing.

Human Error

Approaches to minimising human error include building well designed interfaces and providing non-production sand box environments that users can practice on.

Scalability

The scalability of a system measures how easily we can add hardware to improve performance. Vertical scalability or scaling up involves adding hardware to a single server . Horizontal scalability or scaling out involves adding more servers.

To talk about scalability, we need to be able to measure how the performance of a system is affected by increases in load. To describe load, we need to choose a relevant load parameter.

Typical Load Parameters

- Requests per second
- Ratio of reads to writes
- Hit rate on a cache
- Concurrent users

Once we have decided on the relevant load parameter, we need to look at what happens to performance when the load parameter increases. Typical measures of performance include

Performance Measures

Throughput	How much work can be done per unit of time e.g. bytes per second or transactions per second.
Response Time	Time between sending and receiving a response. Typically we will need to consider average response time.
Latency	Minimum time to get response for trivial piece of logic. Typically, high for RPC

Now that we have load parameters and performance measures, we can make statements such as " System A has a response time of 0.5 seconds with 100 concurrent users and a response time of 1.0 second with 500 concurrent users."

Descriptive

Load Sensitivity	the sensitivity of the performance to increases in the load parameter. .
Efficiency	Performance divided by resources.
Capacity	Upper bound on load before performance becomes unacceptable

Typically, we want to know answer the following questions

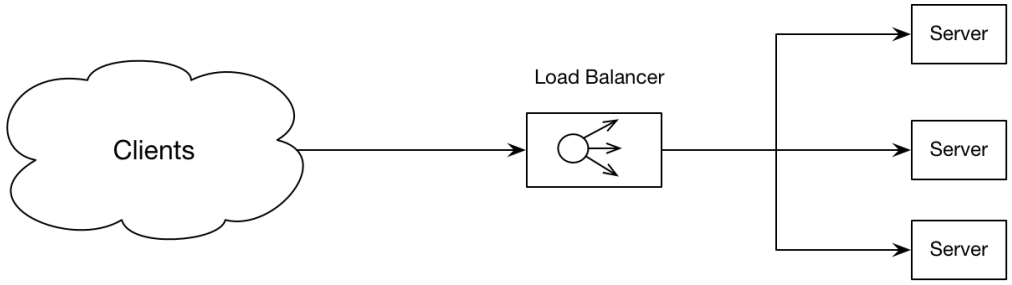
- What happens to performance when we keep system resources constant and increase load?
- If we increase load, how much do we need to increase system resources to keep the current level of performance.

Considerations

Response time averages can hide outliers. Page faults, garbage collection and dropped packets are all causes of increases in response time.

Load Balancing

A special router known as a load balancer routes requests across a cluster of servers to improve system availability, responsiveness and scalability. Load balancers enable horizontal scaling by adding more servers. The load balancer will avoid sending requests to a server that is overloaded, not responding or in error. In this way no server in the cluster becomes a single point of failure.



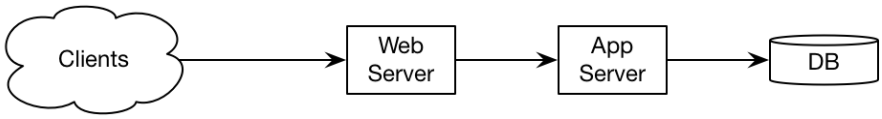
Benefits

The benefits of load balancing are

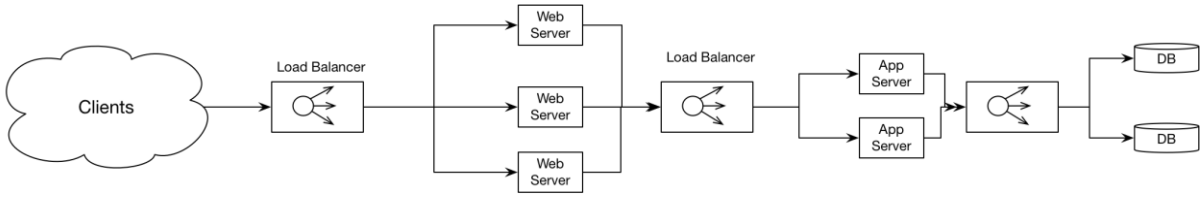
- Faster response time
- Increased availability
- Increased scalability

Example

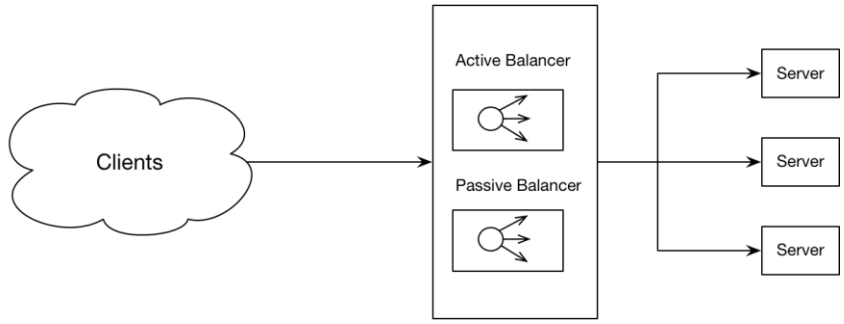
Consider a classic three tier web application



We can maximise scalability and availability by using a load balancer at every tier.



What if the load balancer fails? It is the single point of failure in the system. We can prevent this by using a pair of load balancer that communicate with each other. One can be active and one passive. If the passive load balancer detects the active load balancer is down it can start processing requests.



Routing Algorithms

All the below routing algorithms only choose servers actively processing requests. Heart beating or similar techniques can be used to determine which systems are running.

Round Robin	The simplest algorithm. Can be enhanced by assigning weights to servers. Servers with more capacity get higher weights and get assigned more requests
IP Hashing	Hash the IP address of the client to determine which server to assign to
Least Connections	Assign to the server currently processing the fewest active connections
Lowest Response Time	Assign to the server with the lowest average response time
Lowest Bandwidth	Assign to the server handling the lowest bandwidth.

Caching

Caching involves copying or storing data in fast storage close to the application that will consume (or at least closer than the original data store). Caches load balancers will facilitate horizontal scaling

Caching can provide the following benefits.

- Improved performance and scalability
- Improved availability
- Avoid repeated calculations

Caching is most affective when the client repeatedly reads the same data where the data itself is

- Read more than written
- Relatively static

Often the original data source is

- Far from the application such that latency is an option
- Slower than the cache
- Subject to a lot of contention

Distributed applications use one of two strategies

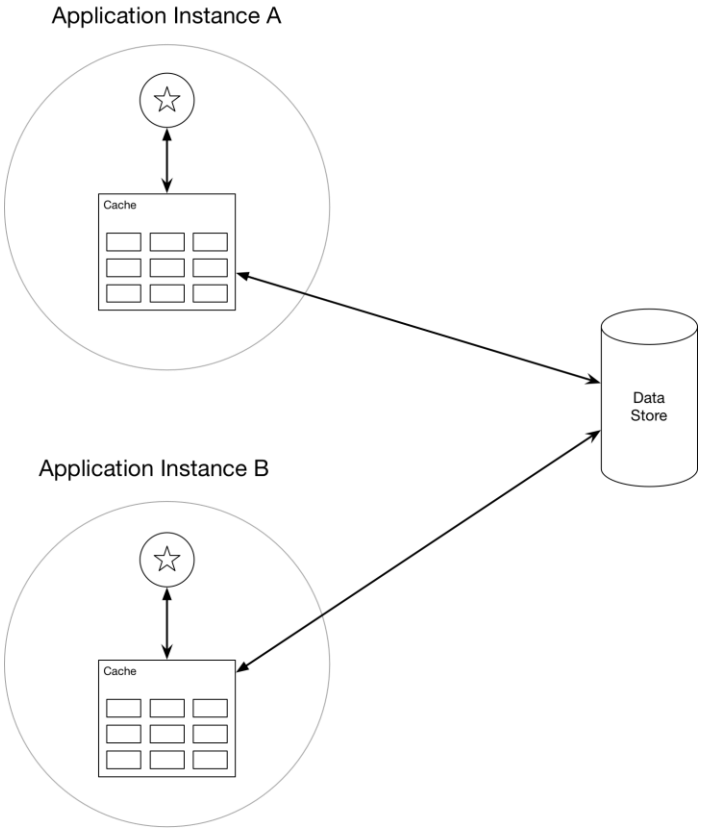
Private Cache and Public caches

In a distributed application caching can be either

1. Private cache – data held on same machine as application accessing it
2. Shared cache – common resource accessed by multiple processes and machines

Private Cache

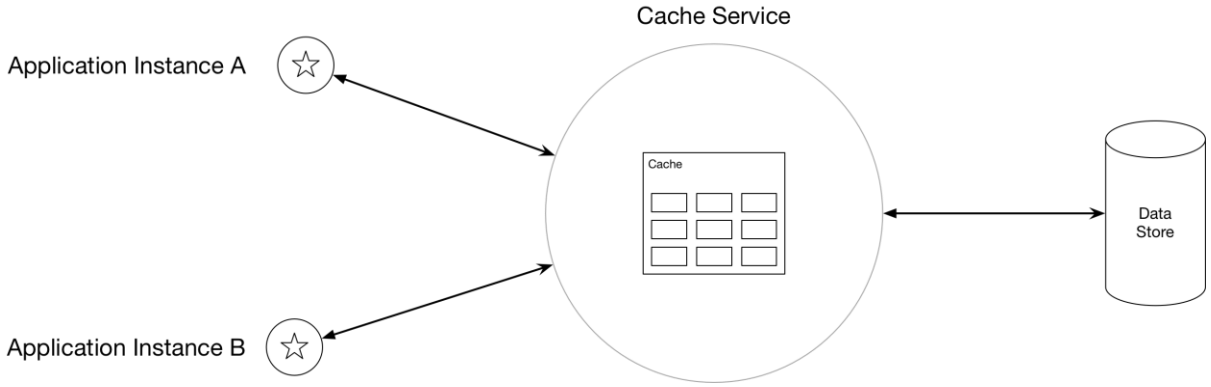
The simplest type of private cache is an in-memory store. It is very fast, however the amount of data that can be cached is constrained by the amount of memory on the machine. If we need to store more than can be accommodated in-memory we can use the local disk. Retrieving data from such a cache will be slower than from an in-memory cache but hopefully faster than accessing a remote data store across a network.



If multiple machines have their own private cache it is likely at some point different app instances will have different versions of a given piece of data in their caches. Shared caching can help with this issue

Shared cache

A shared cache as a service can help prevent the problem of multiple private caches having different versions of the data. Such a service typically uses a cluster of servers onto which it distributes the data. This enables the cache to be scaled by adding more servers.



Stale Data

When the original store changes after the cache has read from it the cache data becomes stale. A caching system supports expire dates such that the period during which data is out of date is minimised. As cached data expired it is removed from the cache and the application must retrieve the data from the original data store. The expiration period can be specified as an absolute time or as a sliding window. Deciding on the best expiration period is a trade-off. If one sets it too short, then the benefit of the cache will be reduced. Too long and the data will be stale more often.

If a cache becomes full the cache must evict data. Typically, it will follow a strategy to decide which data to evict. The default strategy is often LRU, however there exist alternatives

- MRU – assuming it won't be needed again soon
- FIFO
- Explicit removal

Web Clients

A web client caches data using the URI of resources as a key. A web application can then force the client (browser/proxy) to fetch the latest version.

Concurrency

Caches that are shared and updated by multiple application instances have the same concurrency issues as any data store. There are two approaches we can take to concurrency.

- Optimistic concurrency
- Pessimistic concurrency

With optimistic locking the updating application checks to see if the data has changed since it was read. If it has not been changed then it is updated. If the data has been updated since it was read it is up to the application to decide what to do. The optimistic strategy works well when updates are infrequent and there is low risk of collisions.

In pessimistic locking the cache is locked on retrieval and the lock is held until the update is down. Typically, it is useful for short lived operations where collisions are likely. The major downside is that it can limit concurrency.

Cache Down

If the cache goes down there are a number of strategies, we can use. We can fall back to the original store. This can generate a huge amount of traffic on the original store. One alternative is to use a combination of local private cache and remote public cache. If we are using a cache as a service provides they can carryout automatic failover.

Security

We can protect our data by using SSL to protect data over the wire and use authentication and authorisation services on a cache as service.

CDN

Misc.

We can either cache at application start up (seeding) or on demand the first time data is retrieved. Seeding can impose a large load on servers when the application starts up. In order to determine the best time to cache it is necessary to carry out performance testing and usage analysis.

Immutable data or data that changes infrequently lends itself well to caching. Examples would be reference data or product data.

Partitioning

Sharding splits data across multiple data stores in such a way that we can work out which information is on which host. For an in-depth discussion of partitioning see

Azure Best Practices For Data Partitioning

TABLE 1 BENEFITS OF PARTITIONING

Scalability	Dividing data across multiple data stores prevents us being limited by the physical limits of a single store
Performance	Splitting data cross multiple data stores can lead to better performance as we need to search through a smaller amount of data on each partitioned store.
Flexibility	We can allocate different types of data to different types of data store. In this way the data store used is the one most appropriate for the type of data.
Availability	

There are several different ways of partitioning the data.

HORIZONTAL PARTITIONING / SHARDING

Each partition is separate data store. All data stores have the same schema and holds a subset of the data. The following sections describe some strategies for allocating subsets to partitions

Range Strategy

VERTICAL

Each partition holds a subset of the fields. Fields are divided according to how they are used e.g. frequently accessed fields might go into one partition.

Functional

The partitions are determined by the bounded contexts of the architectural solution. We might put orders in one partition and product definitions in another partition.

Key/Hash Based

Given N servers put the data on mod(key,n). As we add servers we need to repartition all the data which is expensive.

Directory based

Use a lookup table to prevent repartitioning as we add servers. The drawback of this approach is that the lookup can become a single source of failure and the extra level of performance can impact performance.

Questions – Systems Design Techniques

Caching

Introduction

What is caching?

Copying or storing data in fast storage close to application

What is the intention of caching?

Improve performance and scalability and availability

Reduce latency and contention associated with large volumes of concurrent reads

Avoid repeated calculations

For what kind of data is caching most effective?

Client repeatedly reads the same data

Data is read more than written

Data is relatively static.

What properties of a data store make caching a good option?

Far from the application such that latency is an issue

Slower than the cache

Subject to a lot of contention

Give example of where cache is particularly useful at improving performance?

DB supports limited number of concurrent connections

Even if this number is exhausted clients can still access data via the cache

Give example of where cache is particularly useful at improving availability?

DB goes down and clients can access data via a cache

Caching Strategies

What are the two caching strategies?

Private – cache held on machine that is running an application

Private - cache shared by multiple machines and processes.

What is the simplest form of private cache?

In memory store

What is the disadvantage of in memory cache?

The amount of data that can be cached is constrained by the machine's memory

How can a private cache overcome this limitation?

By writing to disk

What are the benefits of private cache?

Fast

What are the disadvantages of private caches?

Multiple applications have their own private cache and different versions of the same data

What can be used to overcome this problem?

Use a shared/public cache

Describe the features of cache as a service

Utilises a cluster of servers onto which software distributes the data.

What does this enable?

The cache can be scaled by adding more servers.

What are the disadvantages of shared caches?

Slower than private caches

Increased complexity

What is key to good caching?

Determining what and when to cache.

Stale Data

Describe a general problem with caching?

Data can become stale if the original store changes after cache is updated.

How do caching systems deal with this?

By supporting expire dates

As cached data expires it is removed from the cache.

How can the expiration period be specified?

As an absolute time or as a sliding window.

What are the trade-offs when setting cache expiration period?

To short and objects expire to quick reducing benefit of cache

Too long and data becomes stale more often

What happens if a cache becomes full?

Caches evict data on a LRU basis by default

What are the alternatives to LRU?

MRU – assuming wont be needed again soon

FIFO

Explicit removal such as data being modified

Web Clients

How is client side caching done on a web client?

The URI of a resource is used as a key

The web app can force the client (browser/proxy) to fetch the latest

Concurrency

What are the two forms of concurrency strategy?

Optimistic

Pessimistic

What is optimistic?

Before updating the data app checks if cache has changes since it was read. If not update is made, otherwise app need to decide what to do

When is optimistic useful

Infrequent updates

Low change of collisions

What is pessimistic?

Data is locked in cache on retrieval and cannot be accessed until the update is done.

When should pessimistic be used?

Short lived operations

Collisions are likely

Need to consistently apply multiple updates

What is the downside of pessimistic?

Limit concurrency.

Cache Down

What are options when cache goes down?

Fall back to original data store

Why is this bad?

Huge load on data store

Give alternative

Use combination of local private cache and remote public cache.

Give another alternative

Cache service provides automatic failover

Shared caches distribute data over nodes and rebalance the data.

Security

How can one protect cached data?

Use an SSL connection to protected data over TCP/HTTP

Use an authentication and authorisation model on cache service

Networking

Term	Description
Bandwidth	Maximum amount of data per unit of time e.g. gigabytes per second
Throughput	Actual amount of data per unit of time
Latency	The delay between the sender sending and the receiver reiving information.

We can think of bandwidth as the width of the pipe and latency as the delay incurred by packets traveling from one end of the pipe to the other.

Most people notice a delay of 100-200ms and any delay greater than 300ms is considered sluggish. 1second starts to be a bad delay. To provide a great use experience we need to aim for response times of 100-200ms. This is one of the driving factors behind the adoption of content delivery networks to locate servers close to users.

https://medium.com/@jakob_anderson/speed-performance-and-human-perception-70ae83ea144e

Latency and Bandwidth

"Latency, not bandwidth, is the performance bottleneck for most websites! To understand why we need to understand the mechanics of TCP and HTTP protocols." – High Performance Networking

If I use a high bandwidth, high latency system I would expect to wait a while for my file to start downloading but from the point I see the download start it would be fast. If I was using a low latency, low bandwidth channel the file would start downloading immediately but the download would take a while.

Latency

Latency describes the time taken by a packet to travel from sender to receiver. Latency is impacted by the following factors.

Propagation Delay	Length of channel dividend by the speed of the signal travelling down it. Typically, the speed of the signal is close to the speed of light.
Transmission Delay	Time required to push a packet's bits onto the wire. Simply a function of the number of bits in the packet and the rate of transmission in bits per second. If we have a 100Mb file then it will take 100 seconds to push it onto the wire using a 1Mb/s link and 1 second to push it onto the wire using a 100Mb/s link
Processing Delay	The delay incurred by a router processing the packet header, handing errors and deciding on the next destination
Queuing Delay	If incoming packets arrive faster than a router can deal with them, they get queued inside a buffer

Typically, a packet traveling across a network will encounter many intermediate routers and will hence be impacted by multiple processing and transmission delay. Where load is heavy, we have increased risk of incurring queueing delays.

Very roughly we can assume that a signal in fibre travels at about 200,000,000 metres per second. The distance between London and New York is 5500km so the round trip in seconds is 0.055 seconds or 55ms. The world circumference is 40075 km so the time taken to circumnavigate the globe is then 0.2 seconds or 200ms. In practice the route is not the minimum distance and there will be multiple transmission, processing and queueing delays.

Another problem is that there is a delay between a home and the ISP of between 10ms and 70ms. This is known as the last mile latency.

These might add 100% giving us 400ms to circumvent the globe and 110ms for round trip between New York and London. This latency is affected by the ISP, technology used and time of day.

Bandwidth

A single fibre optic link support multiple channels via wavelength multiplexing. The capacity of a fibre is 171Gb/s so across all channels we get 70Tb/s. A single cable often has four strands of fibre giving a bandwidth of 280Tb/s. The ends of these fibres, however, are connected to much lower bandwidth technologies such as DSL, cable and wireless technologies. The available bandwidth is driven by the weakest link which is typically these low capacity technologies. In 2018 the worlds average download speed was 46Mbs and upload speed was 22Mbs. My own sky gave figures of 16Mbs and upload 0.9Mbs.

The following online resource provides details on networking

https://hpbn.co

Databases

Relational Model Versus Document Model

In a relational database data is organized into relations known as tables. Each relation is a an unordered collection of tuples known as rows.

The driver behind NoSQL databases are

- Need for greater scalability, large datasets, and high write throughput
- Dynamic and expressive data model.

Database Denormalization

Denormalization introduces redundancy into databases to speed up reads by reducing or eliminating joins.

SQL

Data stored in tables. Typically, each table hold a particular kind of entity and the columns hold the properties of that kind of entity. One row in the table represents one entity.

NoSQL

NoSQL databases were designed to provide

- Greater scalability than relational databases
- Support exceptionally large data sets
- Support extremely high throughput

KINDS OF NOSQL

Structure	Redis, Voldemort, Dynamo
Graph	Neo4j, InfiniteGraph
Document	MongoDB
Columnar	Cassandra, HBase

NoSQL versus SQL

First let us look at some of the properties of relational and NoSQL databases

TABLE 2 PROPERTIES

	Relational	NoSQL
Structure	Hard to add columns after data	Columns can be added after data and not all objects need values for all columns
Scalability	Only easily scaled vertically by adding memory and compute. Horizontal scaling hard	Easily supports horizontal scaling. Can be deployed to the cloud and most implementations support horizontal scaling out of the box
Consistency	ACID transactions	No ACID transactions to improve scalability and performance

Comparing the properties of Relational and NoSQL databases gives us an indication of when we might use each. Relational databases are good when

- We need ACID transactional consistency
- Our data schema is well defined and fairly static

In contract we would look to NoSQL databases in situations where

- We need horizontal scalability using the cloud
- Storing large amounts of non-homogenous objects.

Immutability

Immutability is a design property that has two main benefits. Firstly, it makes code easier to reason about. If I pass an immutable object as the parameter to a method, I don't have to look inside that method to see what it does to my object. I know it can't do anything. Secondly, immutable objects do not require synchronization to work correctly in the presence of multiple threads of execution. Before we look at the new Immutable types that .NET supports, we should look at some different kinds of immutability.

TABLE 3 KINDS OF MUTABILITY AND IMMUTABILITY

Mutable	A type whose internal state can be modified in place.
Immutable	A type who internal state cannot be modified in place. Pseudo mutating methods create new objects that share much of the same memory as the original object but with some new memory that describes the change
Freezable	A type that can be mutated until such time as it is frozen after which it cannot be changed
Read only	A type through which references cannot mutate the underlying data. Typically, the underlying type can be mutated via other references

Another benefit of immutable collections is that we get a record of state changes. When we change a mutable collection, the original state is lost.

Immutable collections are also useful when we have snapshot semantics. If we want some threads to see all or none of a batch of modifications. If we were to use a concurrent collection or basic collection, we would need to use a lock and copy the whole collection

Technique

1. Determine the use cases and requirements

   At this stage we want to establish the requirements and use cases of the system we are building. We need to establish the functional and non-functional requirements of the system. There is not one correct solution to these kinds of problem so it is very important to ask enough questions at the requirements stage to scope the solution

2. List assumptions

   Agree with your interviewer reasonable assumptions such as number of requests per second.

3. Capacity and Constraints

   It is worth considering capacity and constraints. At this state we can interested in several measures that will impact our design.

   - Reads per second
   - Writes per second
   - Bandwidth (Reads/Writes per second in bytes)
   - Storage Estimates
   - Memory/Cache estimates

   The results of this section are important later on when we need to consider scalability, partitioning, caching and load balancing

4. Define the System level API

   Defining the API to the system will give us the chance to clarify with our interviewer that we are on the correct lines.

5. Design the database

   At this stage it is worth jotting down some database tables that will hold our data. It is worth considering the type of database technology. Consider when we might use a relational database and why we might use a NoSQL database. By identifying the different data entities and their relationships.

6. Diagram the components (High level design)

   Show boxes etc for major components such as servers and data stores. Once this is done show how data and requests flow through the components to satisfy the requests from

part 2. It is acceptable at this stage to not worry too much about details such as scalability.

7. Design the components (detailed design)

Often, we will not deep dive into every aspect. Ideally the interviewer would help decide which aspects are most deserving of further consideration Topics that might be relevant include

- Application Layer
- Database storage
- Data Partitioning and replication
- Caching
- Load Balancing
- Security