System Design Cheatsheet

> Picking the right architecture = Picking the right battles + Managing trade-offs

Basic Steps

1. **Clarify and agree on the scope of the system**

- **User cases** (description of sequences of events that, taken together, lead to a system doing something useful)
    - Who is going to use it?
    - How are they going to use it?
- **Constraints**
    - Mainly identify **traffic and data handling** constraints at scale.
    - Scale of the system such as requests per second, requests types, data written per second, data read per second)
    - Special system requirements such as multi-threading, read or write oriented.

2. **High level architecture design (Abstract design)**

   Sketch the important components and connections between them, but don't go into some details.
    - Application service layer (serves the requests)
    - List different services required.
    - Data Storage layer
    - eg. Usually a scalable system includes webserver (load balancer), service (service partition), database (master/slave database cluster) and caching systems.

3. **Component Design**

- Component + specific **APIs** required for each of them.
- **Object oriented design** for functionalities.
    - Map features to modules: One scenario for one module.
    - Consider the relationships among modules:
        - Certain functions must have unique instance (Singletons)
        - Core object can be made up of many other objects (composition).
        - One object is another object (inheritance)
- **Database schema design.**

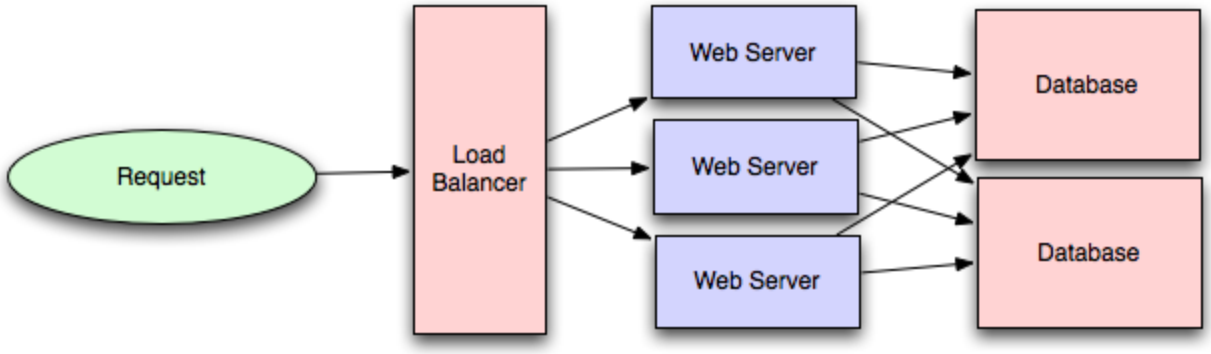
4. **Understanding Bottlenecks**

- Perhaps your system needs a load balancer and many machines behind it to handle the user requests. * Or maybe the data is so huge that you need to distribute your database on multiple machines. What are some of the downsides that occur from doing that?
- Is the database too slow and does it need some in-memory caching?

5. **Scaling** your abstract design

- **Vertical scaling**
  You scale by adding more power (CPU, RAM) to your existing machine.
- **Horizontal scaling**
  You scale by adding more machines into your pool of resources.
- **Caching**
    - Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have, as well as making otherwise unattainable product requirements feasible.
    - **Application caching** requires explicit integration in the application code itself. Usually it will check if a value is in the cache; if not, retrieve the value from the database.
    - **Database caching** tends to be "free". When you flip your database on, you're going to get some level of default configuration which will provide some degree of caching and performance. Those initial settings will be optimized for a generic usecase, and by tweaking them to your system's access patterns you can generally squeeze a great deal of performance improvement.

  - **In-memory caches** are most potent in terms of raw performance. This is because they store their entire set of data in memory and accesses to RAM are orders of magnitude faster than those to disk. eg. Memcached or Redis.
  - eg. Precalculating results (e.g. the number of visits from each referring domain for the previous day),
  - eg. Pre-generating expensive indexes (e.g. suggested stories based on a user's click history)
  - eg. Storing copies of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL.
- **Load balancing**
  - Public servers of a scalable web service are hidden behind a load balancer. This load balancer evenly distributes load (requests from your users) onto your group/cluster of application servers.
  - Types: Smart client (hard to get it perfect), Hardware load balancers ($$$ but reliable), Software load balancers (hybrid - works for most systems)

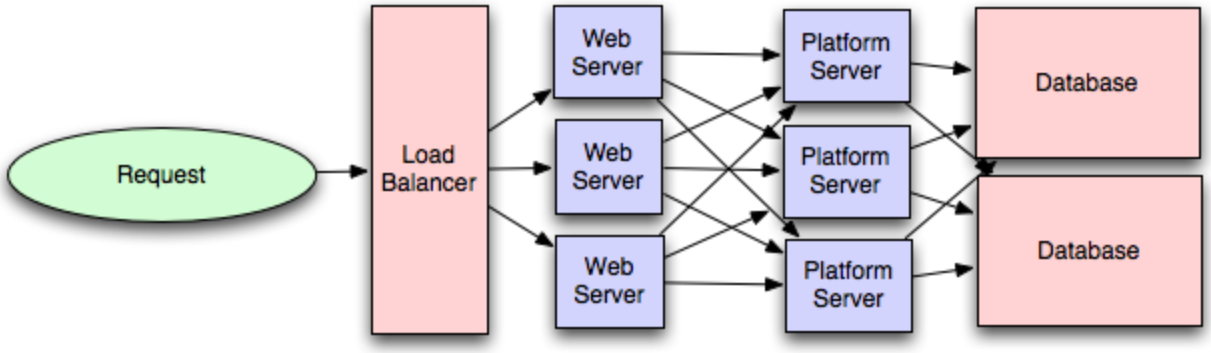


- **Database replication**

  Database replication is the frequent electronic copying data from a database in one computer or server to a database in another so that all users share the same level of information. The result is a distributed database in which users can access data relevant to their tasks without interfering with the work of others. The implementation of database replication for the purpose of eliminating data ambiguity or inconsistency among users is known as normalization.
- **Database partitioning**

  Partitioning of relational data usually refers to decomposing your tables either row-wise (horizontally) or column-wise (vertically).
- **Map-Reduce**

    - For sufficiently small systems you can often get away with adhoc queries on a SQL database, but that approach may not scale up trivially once the quantity of data stored or write-load requires sharding your database, and will usually require dedicated slaves for the purpose of performing these queries (at which point, maybe you'd rather use a system designed for analyzing large quantities of data, rather than fighting your database).
    - Adding a map-reduce layer makes it possible to perform data and/or processing intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports. eg. Hadoop, and maybe Hive or HBase.
- **Platform Layer (Services)**
    - Separating the platform and web application allow you to scale the pieces independently. If you add a new API, you can add platform servers without adding unnecessary capacity for your web application tier.
    - Adding a platform layer can be a way to reuse your infrastructure for multiple products or interfaces (a web application, an API, an iPhone app, etc) without writing too much redundant boilerplate code for dealing with caches, databases, etc.



Key topics for designing a system

1. **Concurrency**

   Do you understand threads, deadlock, and starvation? Do you know how to parallelize algorithms? Do you understand consistency and coherence?

2. **Networking**

Do you roughly understand IPC and TCP/IP? Do you know the difference between throughput and latency, and when each is the relevant factor?

3. **Abstraction**

   You should understand the systems you're building upon. Do you know roughly how an OS, file system, and database work? Do you know about the various levels of caching in a modern OS?

4. **Real-World Performance**

   You should be familiar with the speed of everything your computer can do, including the relative performance of RAM, disk, SSD and your network.

5. **Estimation**

   Estimation, especially in the form of a back-of-the-envelope calculation, is important because it helps you narrow down the list of possible solutions to only the ones that are feasible. Then you have only a few prototypes or micro-benchmarks to write.

6. **Availability & Reliability**

   Are you thinking about how things can fail, especially in a distributed environment? Do know how to design a system to cope with network failures? Do you understand durability?

Web App System design considerations:

- Security (CORS)
- Using CDN
    - A content delivery network (CDN) is a system of distributed servers (network) that deliver webpages and other Web content to a user based on the geographic locations of the user, the origin of the webpage and a content delivery server.
    - This service is effective in speeding the delivery of content of websites with high traffic and websites that have global reach. The closer the CDN server is to the user geographically, the faster the content will be delivered to the user.
    - CDNs also provide protection from large surges in traffic.

- Full Text Search
  Using Sphinx/Lucene/Solr - which achieve fast search responses because, instead of searching the text directly, it searches an index instead.
- Offline support/Progressive enhancement
  Service Workers
- Web Workers
- Server Side rendering
- Asynchronous loading of assets (Lazy load items)
- Minimizing netwrok requests (Http2 + bundling/sprites etc)
- Developer productivity/Tooling
- Accessibility
- Internationalization
- Responsive design
- Browser compatibility

Working Components of Front-end Architecture

- Code
  - HTML5/WAI-ARIA
  - CSS/Sass Code standards and organization
  - Object-Oriented approach (how do objects break down and get put together)
  - JS frameworks/organization/performance optimization techniques
  - Asset Delivery - Front-end Ops
- Documentation
  - Onboarding Docs
  - Styleguide/Pattern Library
  - Architecture Diagrams (code flow, tool chain)
- Testing
  - Performance Testing
  - Visual Regression
  - Unit Testing
  - End-to-End Testing
- Process

- Git Workflow
- Dependency Management (npm, Bundler, Bower)
- Build Systems (Grunt/Gulp)
- Deploy Process
- Continuous Integration (Travis CI, Jenkins)

Links

How to rock a systems design interview

System Design Interviewing

Scalability for Dummies

Introduction to Architecting Systems for Scale

Scalable System Design Patterns

Scalable Web Architecture and Distributed Systems

What is the best way to design a web site to be highly scalable?

How web works?