System Design

Because there is always work to be done

Agenda

- Characteristics of Systems
- Load Balancing
- Caching
- Data Partitioning
- Database Replication
- Queues
- Throttling
- Microservices
- The Secret Sauce of Scaling
- Detecting When a System is Going to Fail

Designing Large Scale Systems

- **Managing risk is a primary concern of system design**
  - Trade offs:
    - Space
    - Time
    - Developer time
      - Building new (parts) of the system
      - Refactoring/technical debt
    - Budget
    - User experience vs. developer experience

Designing Large Scale Systems

- **We have to ask ourselves a few things:**
  - What are the moving parts of system and the different architectural pieces that can be used?
  - How do/will they interact with each other?
  - How can we best use these pieces? What are the risks?

Some Characteristics of Systems

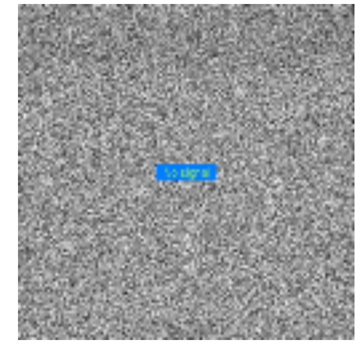
- **Scalability: is our system capable of managing increased demand?**
  - Horizontal Scaling: add more servers (adding more buildings to a neighborhood)
  - Vertical scaling: add more power/resources to a server (CPU, Storage) [add more floors to a building]

Some Characteristics of System

- **Reliability: What is the probably our system is going to fail at any given time period? Hint: Probably very high.**

Some Characteristics of System

- **Availability: How long can our system remain operational to perform whatever it needs to at any given time period?**
  - Ex: You can drive your car for a while without having to do so many scheduled maintenances (oil change, fix brake lining, change various parts, etc.)
  - Reliability vs. Availability
    - If a system is reliable, it is available. But if it's available, it's not necessary reliable.
      - Ex: The recent security breaches of many companies.



Some Characteristics of System

- **Efficiency: How much time does it take to do something?**
  - Latency: Response time in doing something
  - Throughput: How much volume of a thing we need to do in a given time frame
  - Latency vs. Throughput Ex:
    - An assembly line is manufacturing cars. It takes 8 hours to manufacture one car and the factory produces 120 cars per day
      - Latency: 8 hours
      - Throughput: 120 cars per day / 5 cars per hour

Some Characteristics of System

- **Serviceability: How easy is it to maintain or repair our system?**
  - How easy is it to diagnose problems?
  - How easy is it to make upgrades to the system?

Some Characteristics of System

- **Affordability: Do we have even have the money?**
  - Licensing fees
  - Cost of hardware
  - Cost of support
  - Cost of maintenance

Load Balancing

Load Balancing

- **Share the love of traffic/requests across cluster of servers**
  - Improves availability and latency
- **Keeps track of the status of resources**
  - Performs regular health checks; if it fails, the server is removed from the pool
  - If a server is getting hit repeatedly, LB will send request to another server

(Some) Load Balancing Algorithms

- **Least Connection Method**
  - Directs traffic to server with fewest active connections
- **Least Response Time Method**
  - Directs traffic to server with fewest active connections and lowest average response time
- **Least Bandwidth Method**
  - Directs traffic to server with least amount of traffic (in Mbps)
- **Round Robin Method**
  - Cycles through list of servers and sends each new request to a new server
- **Weighted Round Robin Method**
  - Attach a weight to each server that has different processing capabilities

Multiple Load Balancers

- Typically, you want to avoid central points of failure so you might want to add another load balancer to make a cluster of load balancers with one being active and one being passive; that way if one fails, you have a back up

Pitfalls

- Can be a bottleneck source because it may not have enough resources/the algorithm configuration is not working out
- Only one load balancer is a single source of failure
- Multiple load balancers increase the complexity of the application

Caching

Caching

- **The principle of locality: the tendency to access something repeatedly over a short period of time**
  - Temporal locality: need to use a specific set of data over and over again in a shorter time frame
  - Spatial locality: use of data that is stored closely
    - Sequential locality: data that is arranged linearly (ex: array)
- **A cache is like short term memory**
- **Has limited amount of space**
- **Faster than the golden source**
- **Load balancing helps us scale horizontally across an increasing number of servers but caching helps us make better usage of the resources we already have**

Caching

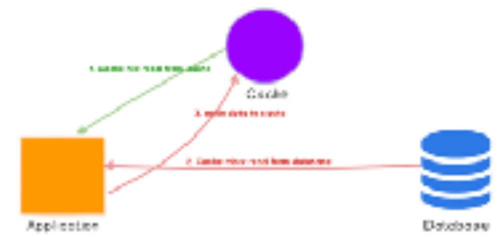
- Application caching
- CDN caching
- Client caching (browser)
- Database caching

Caching Pitfalls

- **Has limited amount of space**
- **Need to maintain data consistency**
- **Need strategies to:**
  - Make sure data is consistent
  - Remove data from the cache as it gets full

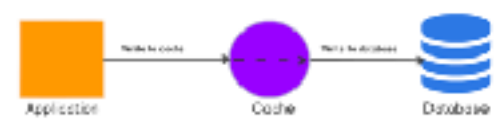
Cache Update Strategy: Cache Aside

- **Cache Aside**
  - The application is responsible for reading and writing from storage; the cache doesn't interact with storage directly



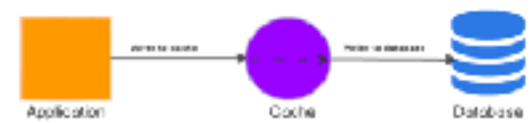
Cache Update Strategy: Write Through

- **Write Through**
  - The cache is the main data store for the app
  - The cache is responsible for reading and writing to the DB
  - The writes are done at virtually the same time



Cache Update Strategy: Write Back

- **Write Back**
  - Add/update entry in the cache
  - Asynchronously write entry to the DB under specified intervals/ conditions



Cache Eviction Policy: Least Recently Used

- **Least Recently Used**
  - Discards the least recently used items
  - Case Study: File suggestion at dropbox

Cache Eviction Policy: Least Frequently Used

- **Least Frequently Used**
  - Counts how often and item is used/needed, then discards the ones that are used the least
  - Case Study: Mobile keyboard suggestions

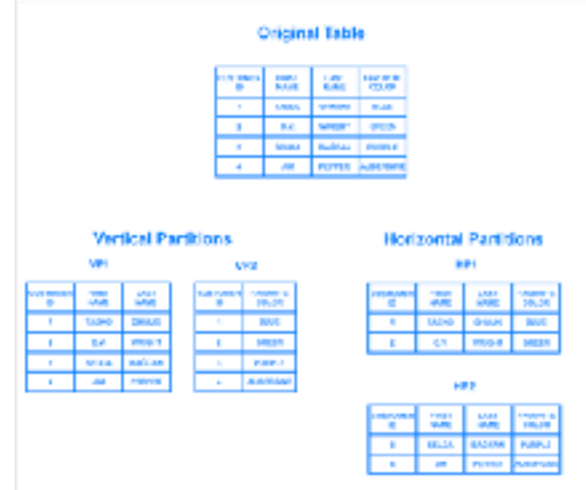
Cache Eviction Policy: Most Recently Used

- **Most Recently Used**
  - Remove the most recently used items
  - Case Study: Tinder swipe suggestions

Sharding/Data Partitioning

Sharding/Data Partitioning

- **Breaking up a huge DB into many smaller parts and distributing them across multiple servers.**
  - Callback to scaling vertically vs. horizontally: After a certain point it's cheaper to scale horizontally



Partitioning Methods

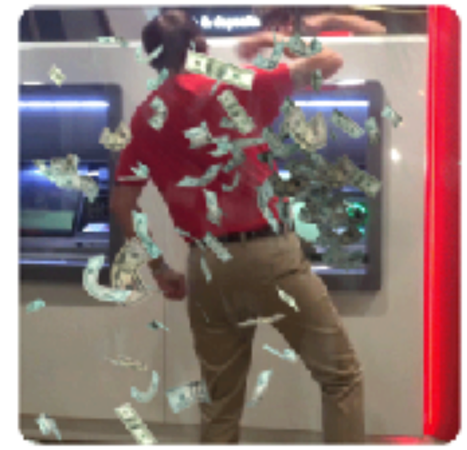
- **Horizontal partitioning (Sharding)**
  - Put different rows into different tables
    - Think: If we are storing different areas into a table, we can decide that locations with ZIP codes over 50000 will go in one table and those that are under 50000 go into another
- **Vertical partitioning**
  - Divide our data to store tables related to a specific feature in our application into their own server (more tables; fewer columns)
    - Think: If we are building Instagram and trying to store data on users, photos they upload, and people they follow, we can place profile information in one server, photos on another, and follows in yet another

Database Replication

Because we need our data to be stored EVERYWHERE

Case Study: ATM or Unlimited Money

- Withdraw $1000 from an ATM
- The transaction is sent to a database
- But what happens if something bad happens like a network failure in that town? Unlimited money?
- Sadly no, the bank probably has another database in another town



Queues

Wait… Haven't we seen this before?

Message Queuing

- **Asynchronous service-to-service communication**
  - Programs/services communicate by sending each other data messages instead of calling each other directly
    - Sounds very familiar…
- **Messages (data) are put on a queue until they are processed**

Case Study: (Early) WhatsApp

- **XMPP Message Standard**
  - Built their own internal messaging system as of 2015
- **Queues… queues everywhere**
- **In fact, one of their primary gauges of system health was monitoring message queue length**

Throttling/Rate Limiting

Protect Your API

- **One of the biggest benefits of throttling is to protect your API from a barrage of requests**
  - Affordability
  - Availability
- **All those APIs you use that prevent you from requesting too many times and will block you (or charge you) if you do?**

(Some) Throttling Algorithms

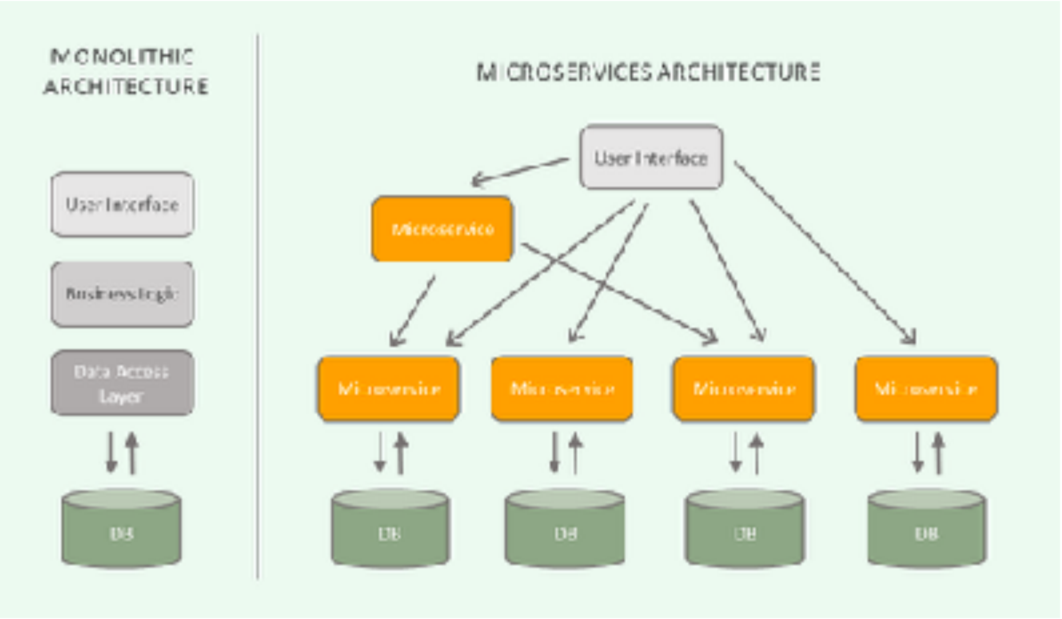
- **Leaky Bucket**
  - Shove requests into a queue and at specified intervals, process the first requests in the queue
- **Fixed Window**
  - Track rate of requests in a fixed interval. Each incoming request increments a counter defined for that window of time. Discard the requests that increment the counter over a specified threshold.

Microservices

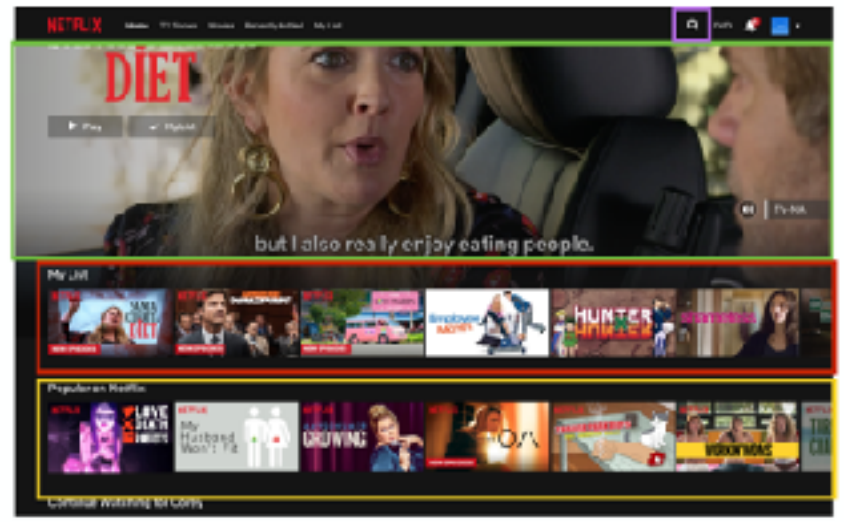
Slaying Goliath (Monoliths)?

Microservices

- **A suite of independent, small, modular services**
  - Highly maintainable*
  - Highly testable
  - Independently deployable
  - Probably organized by business
- **But wait… How do all these microservices communicate?**
  - This happens to be one of its more major drawbacks
- **What are some other drawbacks?**

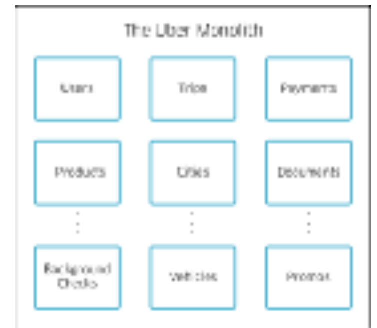

MONOLITHIC ARCHITECTURE
User Interface
Business Logic
Data Access Layer
DB
MICROSERVICES ARCHITECTURE
User Interface
Microservice
Microservice
Microservice
Microservice
Microservice
DB
DB
DB
DB

Microservices



Case Study: Uber

- Started off with a monolithic architecture design
- In 2015, 500 microservices
- As of 2017, 2000+
- Service Oriented Architecture



The Secret Sauce of Scaling

A really closely guarded secret in industry

There is none

Wait.... wut....

How do we know when something is going to fail?

We'd like to avoid epic fails

You Don’t

Wait... wut....

Netflix: Chaos Monkeys

How many monkeys does it take to create (or break) Netflix?