Fullstack for Frontend:
systems design

FS
FE

Fullstack System Design

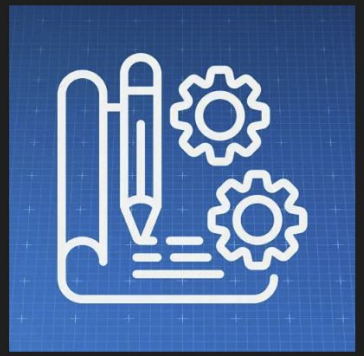
FS
FE

Jem Young

Software Engineering Manager

@jemyoung

Jem Young



Engineering Management Fundamentals 101

Jem Young
Netflix

4 hours, 29 minutes CC

Full Stack for Front-End Engineers, v3

Jem Young
Netflix

8 hours, 12 minutes CC

Web Assembly (Wasm)

Jem Young
Netflix

3 hours, 43 minutes CC

Interviewing for Front-End Engineers

Jem Young
Netflix

2 hours, 53 minutes CC

Course Overview

Foundations

Scoping

Functional

Non-functional

Modeling

Scaling Up

Vertical

Horizontal

Load Balancers

Data Storage Intro

Relational

ACID

Sharding

System Quality

Reliability

Performance

Observability

CAP Theorem

Consistency

Availability

Partition Tolerance

Caching

Client and server

CDN's

Invalidation

Working at scale

Security
Authentication
Authorization
SSL/TLS

Resilience
Edge cases
Handling failure

Data Storage
Non-relational
Replication

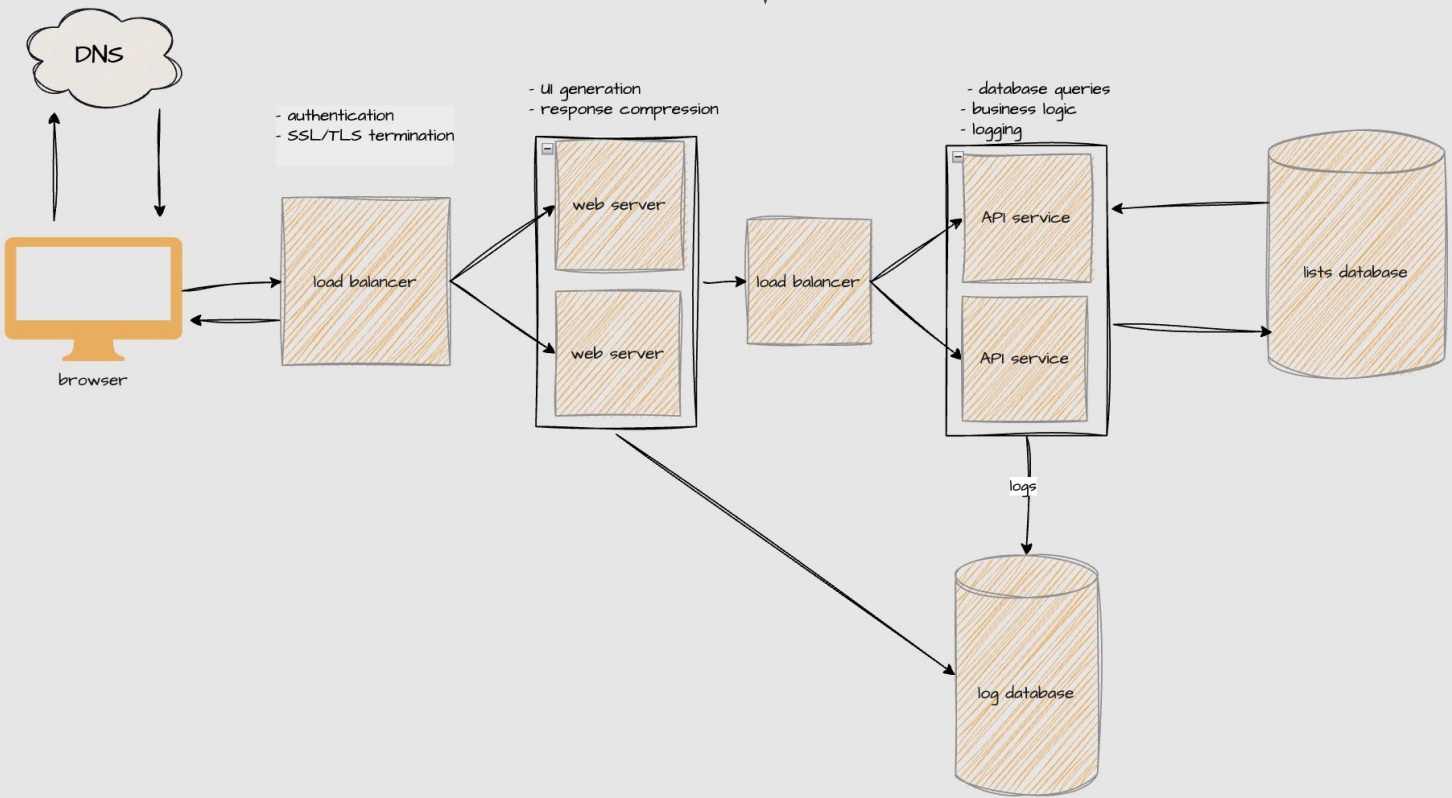
Protocols
HTTP
GraphQL
REST

Async workflows
Queues
Message brokers
Data processors

Case Studies

trade offs

"design a note taking app"

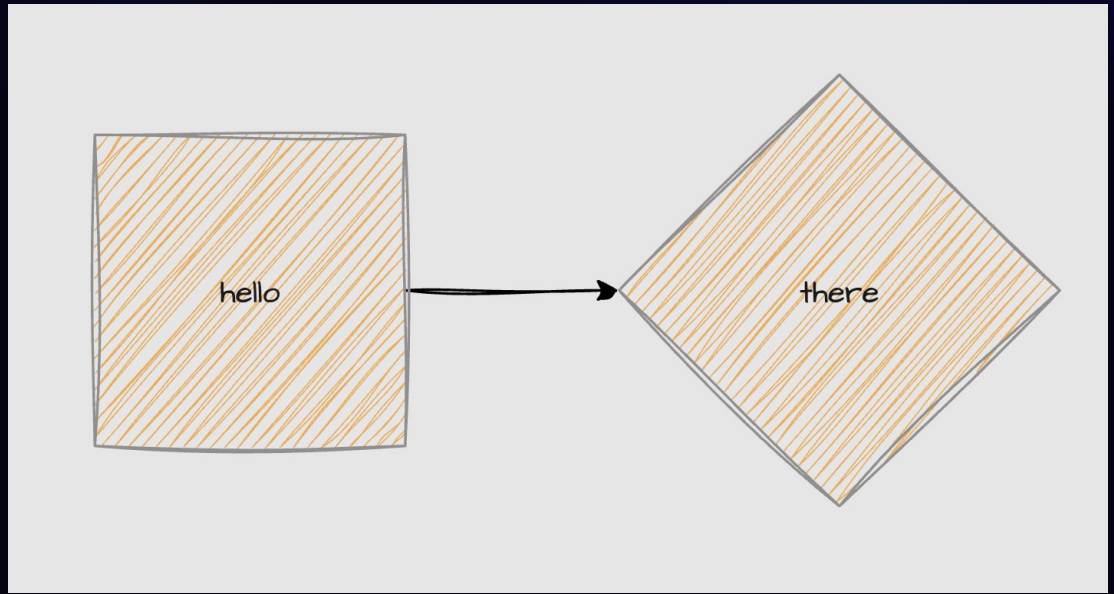


crawl, walk, run

Tools

[Draw.io]

[Excalidraw.com]



systems thinking

Chapter 1

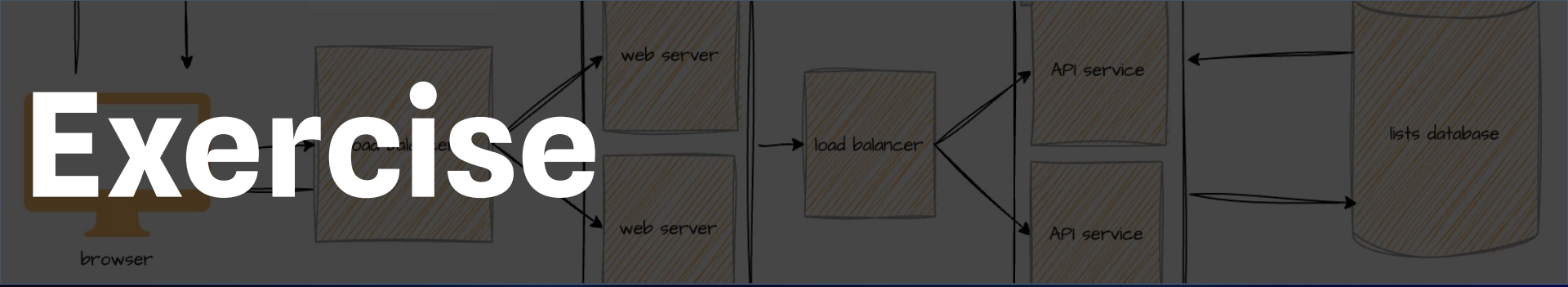
everything is a system

Everything is a system

Everything is a system

What is a system

- A collection of components that work together
- Has inputs and outputs
- Has boundaries (what's inside, what's outside)

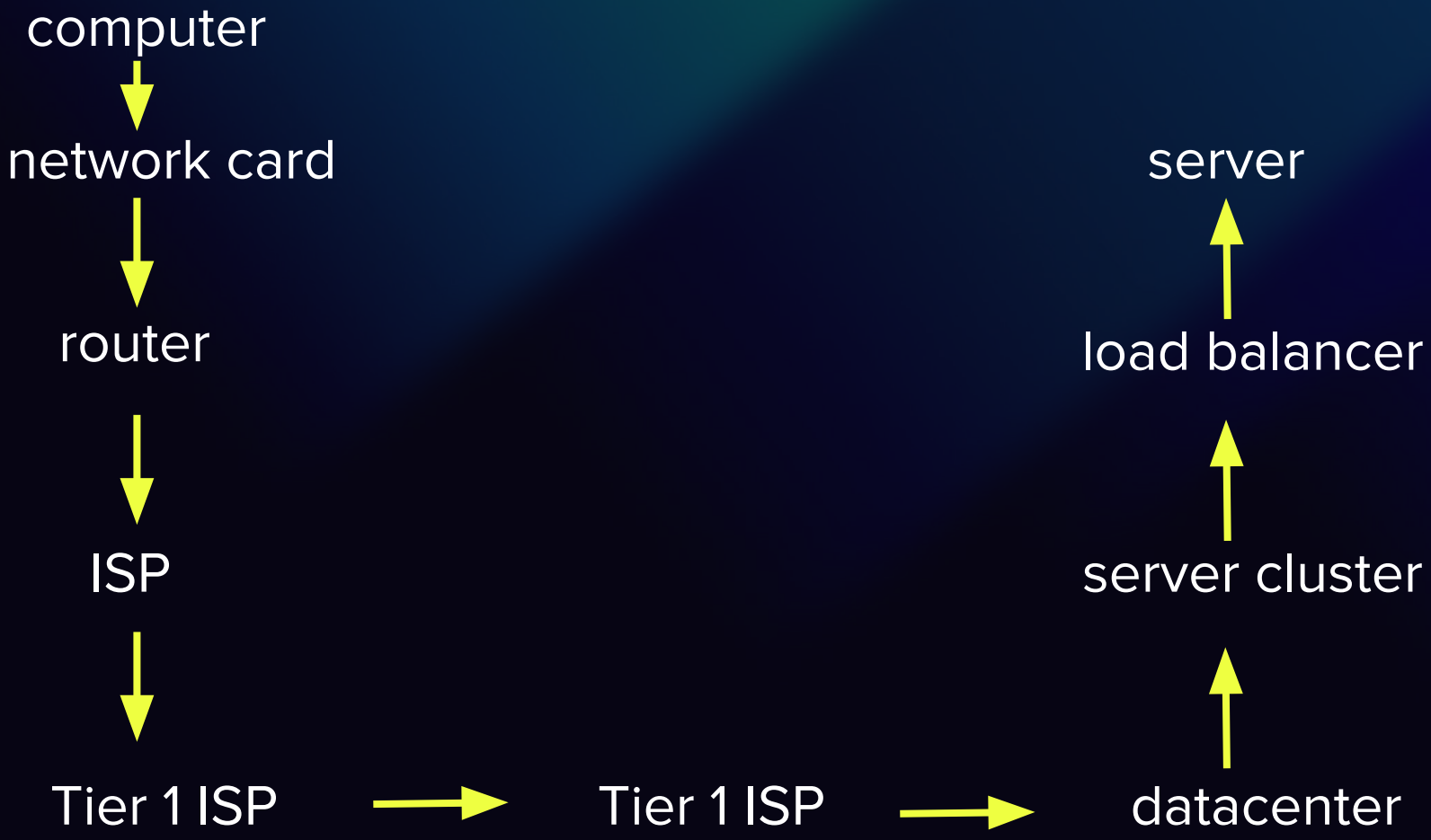


Exercise

Design a pizza shop

What is a *distributed* system

- Multiple computers working together as a single system
- Components may be physically separated
- Designed to handle **failures** and **scale** across multiple locations


computer
network card
router
ISP
Tier 1 ISP
Tier 1 ISP
datacenter
server cluster
load balancer
server

Common System Components

Component	Role
Client	Sends requests, displays data to users
Database	Stores, updates, and retrieves data
Server	Processes requests, business logic
Load Balancer	Distributes incoming traffic to keeps things running smoothly
Cache	Makes things faster by temporarily storing data

client

Example types

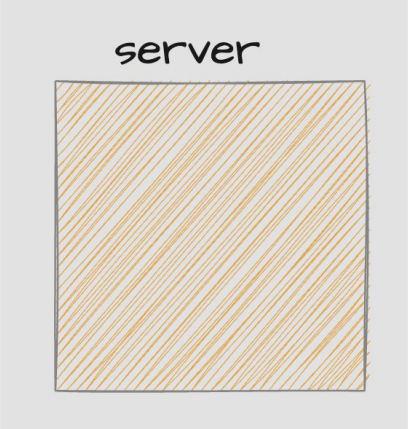
- Browsers
- Mobile apps
- TV's

Inputs

- User actions

Outputs

- Server requests
- UI updates



Example types

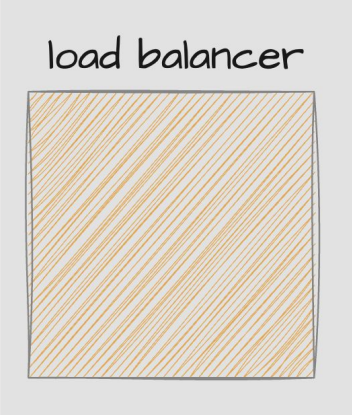
- web server
- API
- video processor

Inputs

- requests

Outputs

- responses (HTML, JSON, etc)
- server requests
- database queries
- modified data



Example types

- software LB
- hardware LB

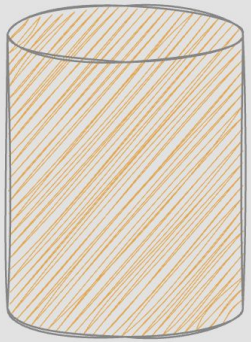
Inputs

- requests

Outputs

- routed requests

database



Examples types

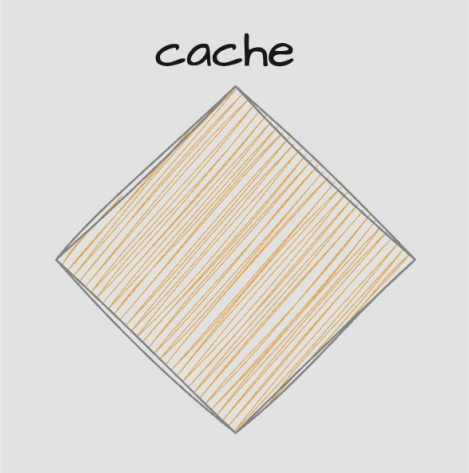
- relational (SQL)
- non-relational (NoSQL)
- specialized (Graph, object-oriented, vector, etc)

Inputs

- queries

Outputs

- data
- status response
- objects (images, etc)



Example types

- client
- server
- CDN's
- in-memory

Inputs

- keys

Outputs

- cached data

message queue

Examples

- point-to-point
- publish/subscribe

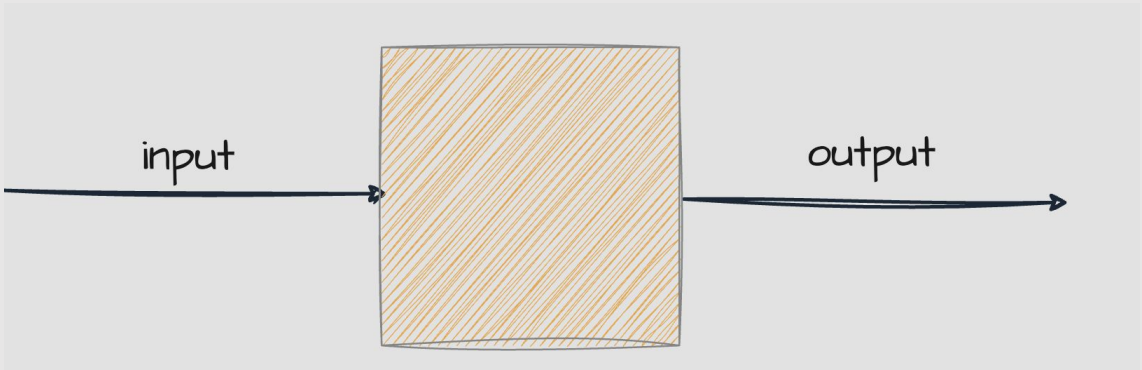
Inputs

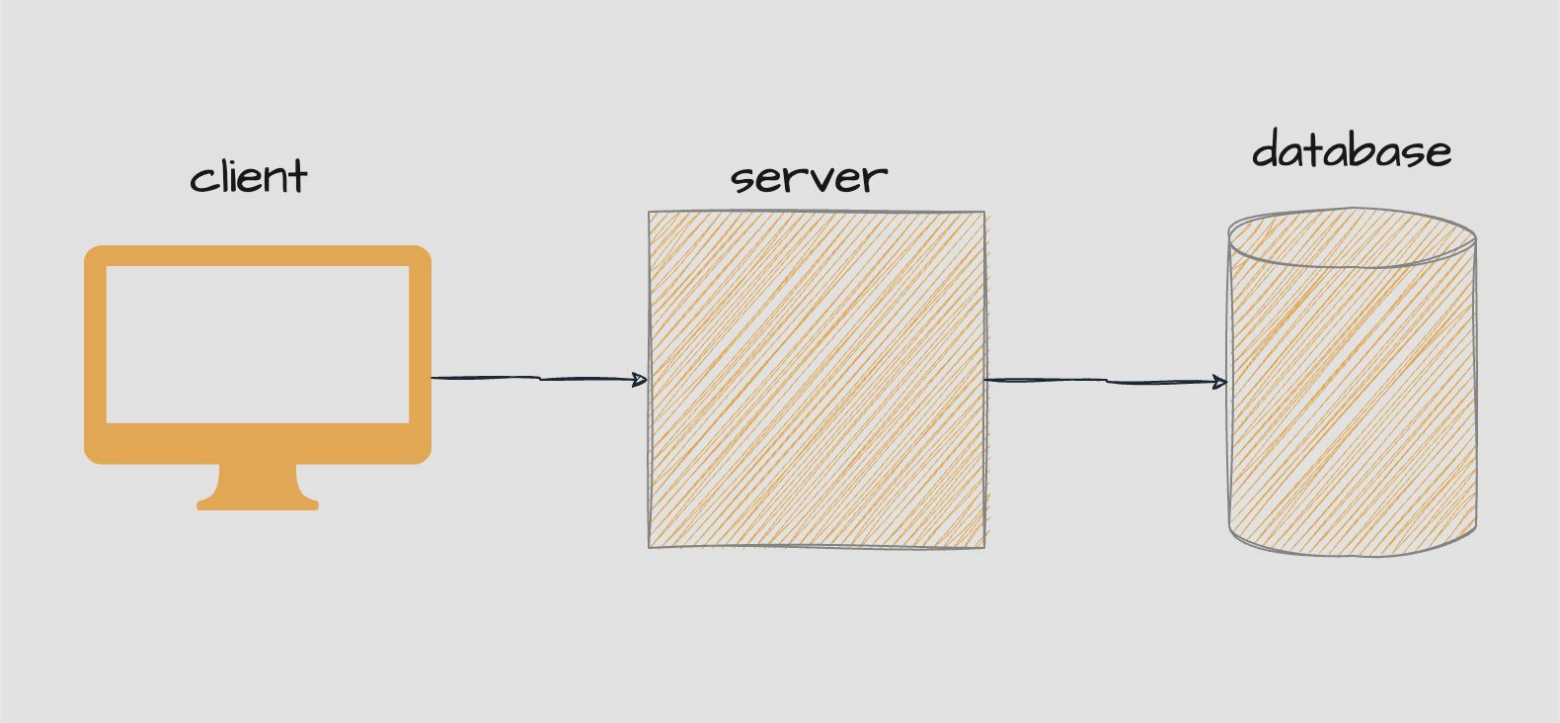
- messages

Outputs

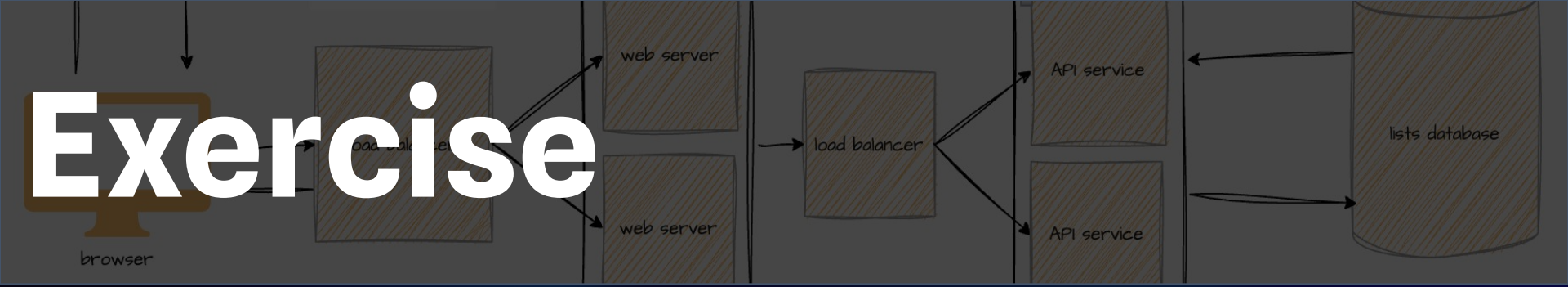
- messages


connection


input
output


client
server
database

Exercise



Visualizing the flow

Summary

everything is a system

- Everything is a system with inputs, outputs, and boundaries
- All distributed systems have common building blocks with specific roles
- Understanding systems helps you see both the individual components and how they all work together

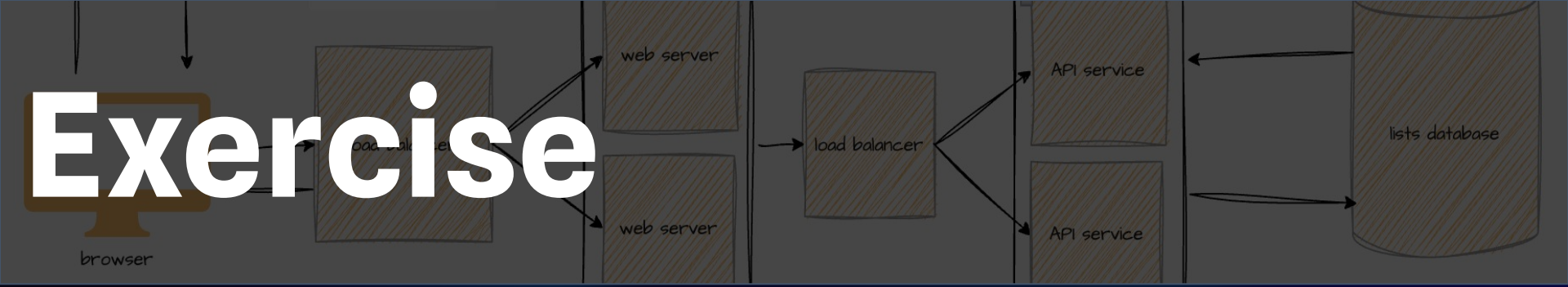
Chapter 2

how to build anything

Core elements of system design

- Translating business requirements
- Designing API and architecture
- Understanding technology and trade offs

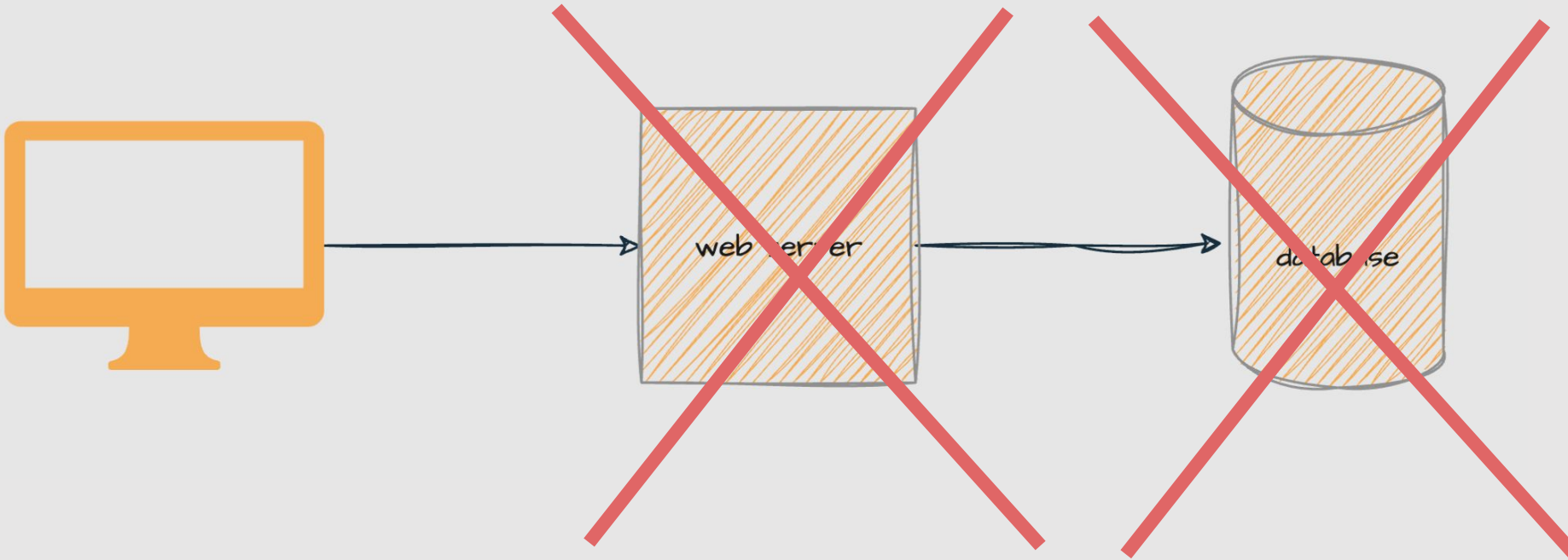
Exercise



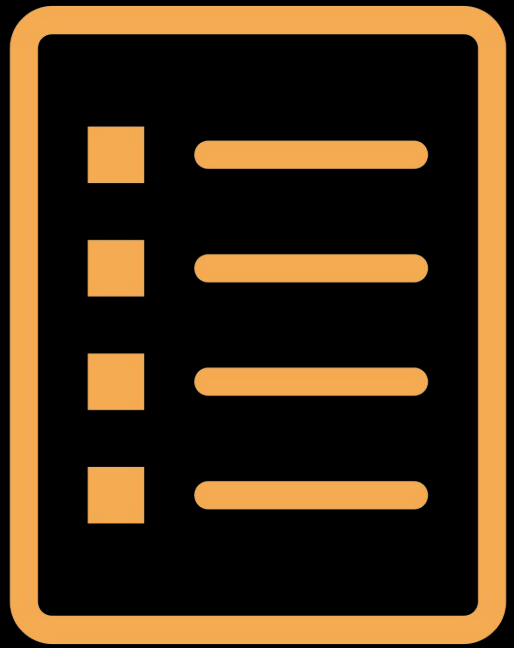
Design a ToDo App

what are the requirements?

- create a todo
- read a todo
- edit a todo
- delete a todo


web server
database

ToDo



What core features do we need to support?

Strategy

1. Scope the problem
2. Design the high-level architecture
3. Address key challenges and tradeoffs

Chapter 3
understanding the problem

Functional Requirements

what are the requirements?

Functional requirements

`Functional`

Describe ***what*** the system should do

`Non-Functional`

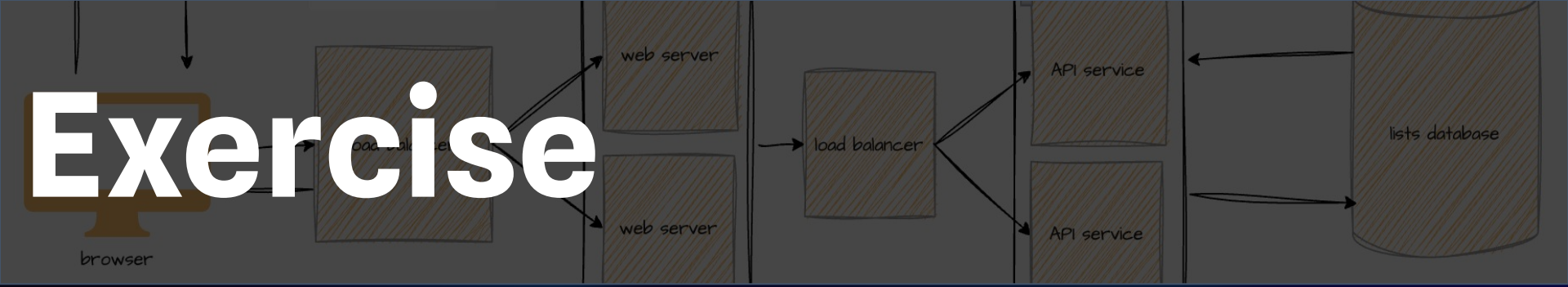
Describe ***how*** the system should perform

Functional requirements

- What are the core features?
  What should users be able to do?
- Who are the users?
  Are there different types of users?
- How do users interact with the system?
  What devices are supported?
- Are there edge cases to consider?
- What are the constraints?

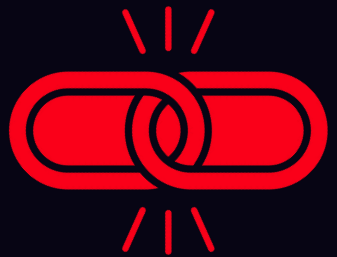
Functional requirements

- User should be able to view the menu of pizzas and toppings
- User should be able to order a pizza
- User can only order 1 pizza at a time
- There is only 3 sizes of pizza
- A pizza can have at most 3 toppings



Exercise

Functional requirements



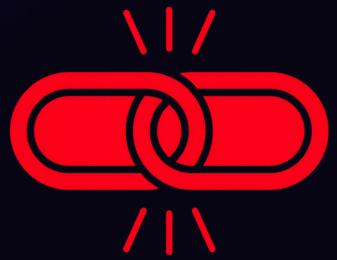
Exercise

Functional requirements

1. Create 3-5 questions to ask
2. Create 5-8 core functional requirements for each

Banking app

A link shortening service



Functional requirements

Create

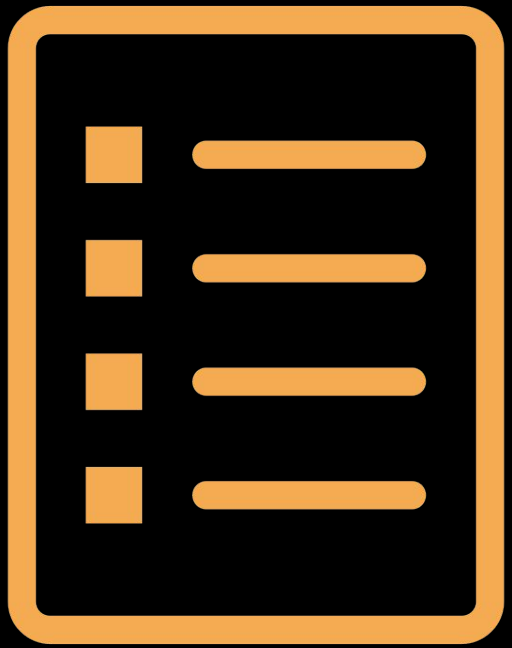
Read

Update

Delete

ToDo

Functional Requirements



Create the functional requirements for the ToDo app

CAP Theorem

Quality

Reliability

Quality

Reliability

The ability for a system to function correctly over time

Availability

The proportion of time a system is operational and accessible

Resiliency

How well does the system handle failures

Consistency

How well does the system ensure that all users see the same data at the same time

CAP Theorem

Consistency

Every read receives the most recent write or error

Availability

Every request receives a (non-error) response

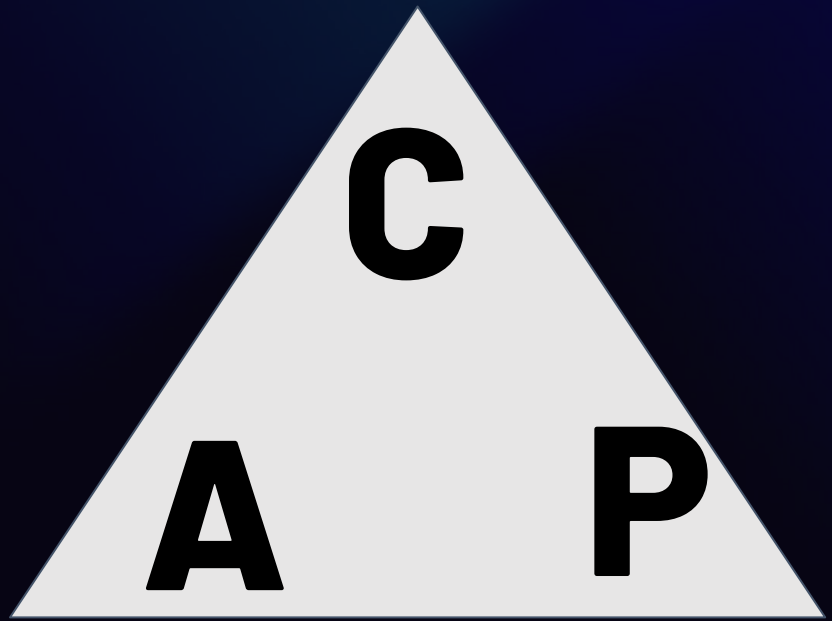
Partition Tolerance

The system continues to operate even if messages are delayed or lost

CAP Theorem

Any distributed system can only **guarantee** 2 out of 3 at the same time

Consistency
Availability
Partition
Tolerance

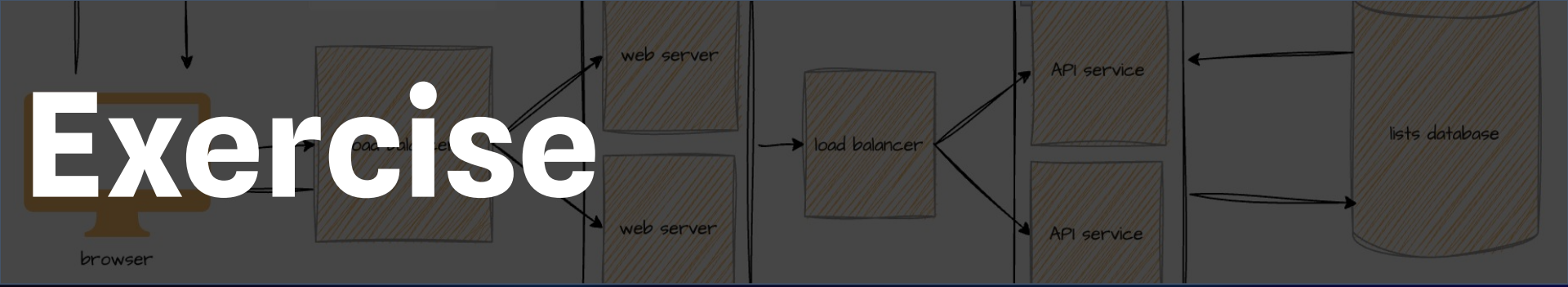


CAP Theorem

trade offs

CAP Theorem

pick two	trade off
~~C + A~~	~~Only works without network issues~~
C + P	Always show the latest data but unreliable performance
A + P	Always responds but might show outdated data



Exercise

Making Trade Offs

Takeaways
CAP Theorem

- ➔ In distributed systems, network failures will happen
- ➔ You must decide: is it more important to have the latest data or to always be available?
- ➔ There's no "perfect" system. Trade-offs are necessary

System Quality

System Quality

- **Reliability**
  - Availability
  - Resiliency
  - Consistency
- Observability
- Security
- Scalability
- Adaptability
- Performance
  - Latency
  - Throughput

System Quality

Observability

The ability to know what is happening in your system

Security

The ability to safeguard your system and its data

System Quality

Scalability

The ability handle increases or decreases in system usage

Adaptability

The ability to handle changing requirements or user behaviors

System Quality

Performanc
e

Latency

How quickly does the system respond

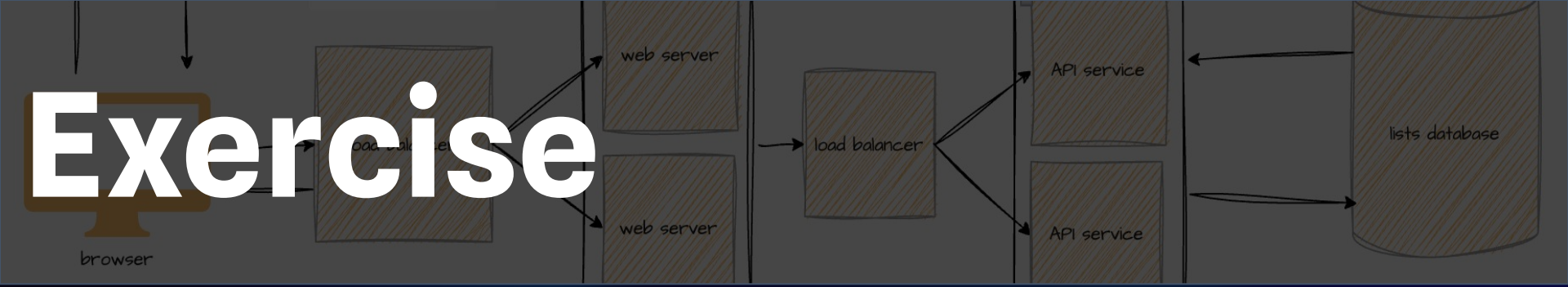
Throughput

How much data can move through the system at a given time

Non-Functional Requirements

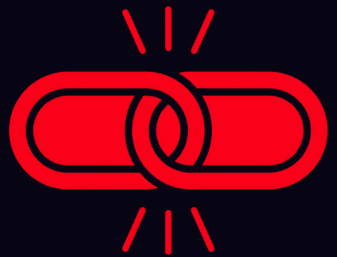
Non-Functional requirements

- How many users do we expect?
  What is the average RPS?
  Does this number change over time?
- How consistent does the system need to be?
  Is it acceptable for users to see slightly outdated data?
- What metrics are important?
  What is the maximum latency?
- What data needs to be protected?



Exercise

Non-Functional requirements

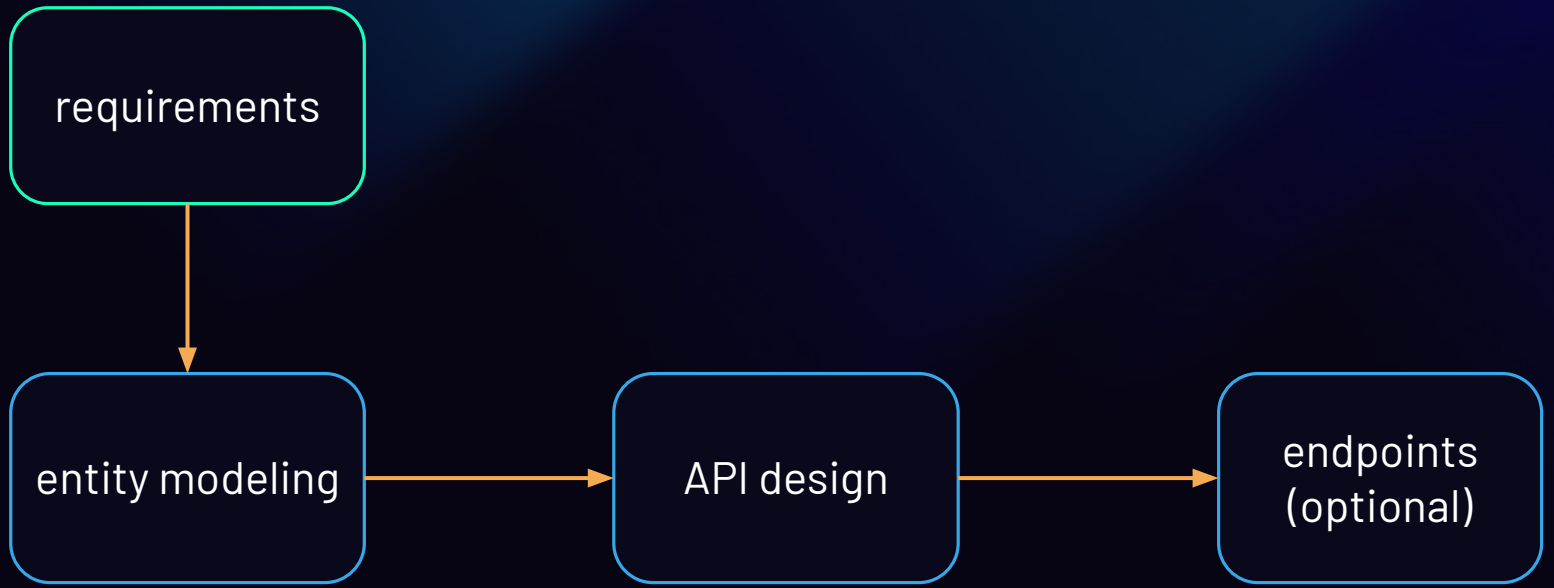


Chapter 4
high-level design

High-level design

1. Modeling
2. Architectural Design

Modeling


requirements
entity modeling
API design
endpoints
(optional)

Modeling

Entity modeling

Define the main functional elements

API Design

Define the actions and operations of the system

Protocols

Protocols

HTTP

Key Characteristics

- simple
- human readable
- supported by all browsers
- stateless

Common use cases

- web browsers

HyperText Transport Protocol

Protocols

WebSockets

Key Characteristics

- bi-directional communication
- persistent connection
- low-latency
- stateful

Common use cases

- chat applications
- Live dashboards
- Collaborative editing

Protocols

Server-Sent Events

Key Characteristics

- one-way communication (server to client)
- human readable
-

Common use cases

- News feeds
- Status updates
- Stock tickers

Protocols

gRPC

Key Characteristics

- binary protocol (HTTP/2)
- Strongly-typed contracts (proto buffs)
- requires code generation

Common use cases

- Microservice communication
- Performance critical systems
- IoT devices

Remote **P**rocedure **C**all

Protocols

REST

Key Characteristics

- multiple endpoints
- human readable
- supported by all browsers
- stateless

Common use cases

- single-sources of data
- CRUD apps
- easily cached data

REpresentational **S**tate **T**ransfer

Protocols

GraphQL

Key Characteristics

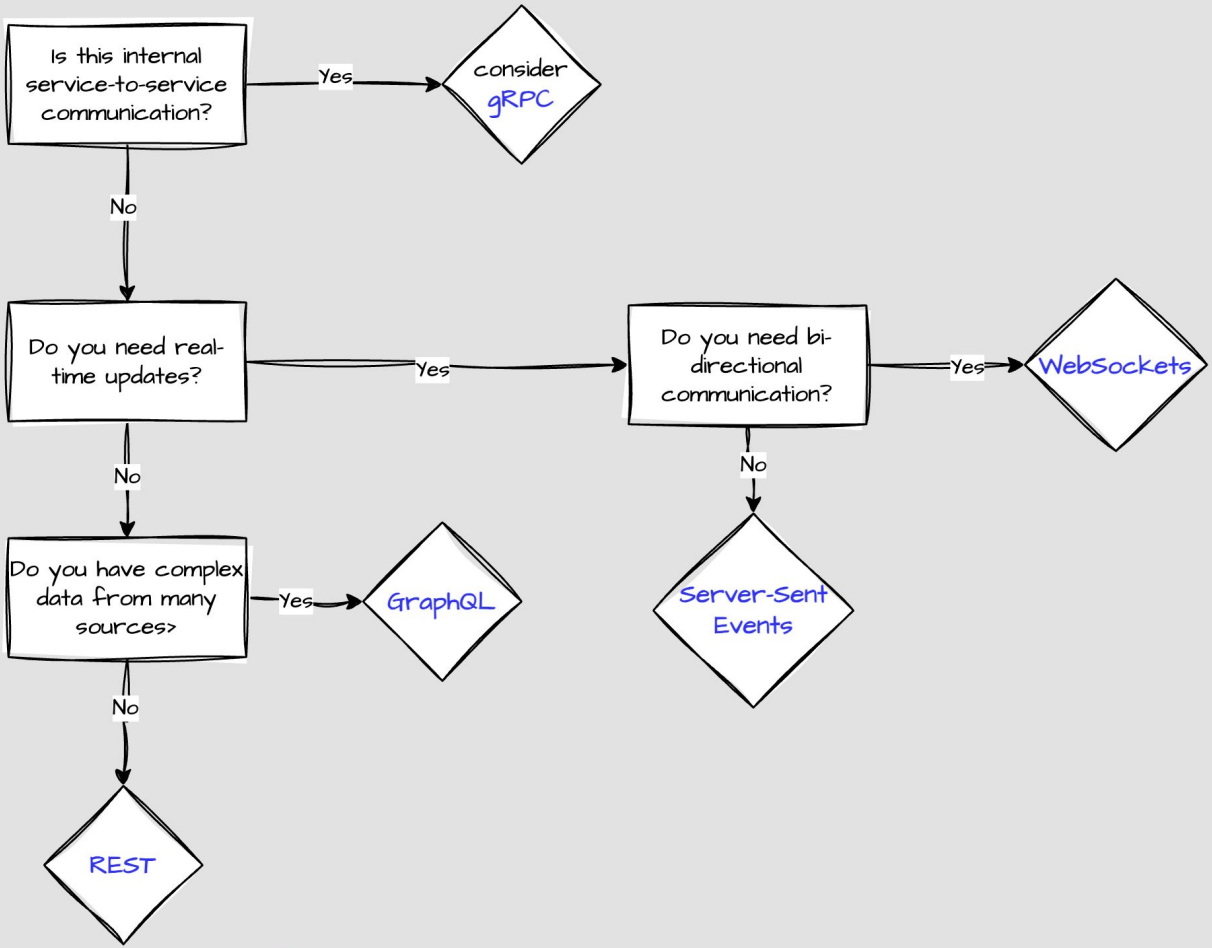
- single endpoint
- precise data retrieval
- self-documenting API
- strongly typed

Common use cases

- complex or multiple sources of data (Facebook homepage)
- apps supporting multiple client types
- decoupling frontend from backend development

Graph Query Language

protocol cheat sheet*



*General advice. Not intended to be prescriptive

Protocols

Case Studies

1. A stock trading dashboard showing real-time price updates
2. A document editor where multiple users collaborate simultaneously
3. A food delivery app that needs to show the delivery driver's location
4. An API for a banking system used by multiple client applications
5. Communication between microservices in a video processing pipeline

High-level design

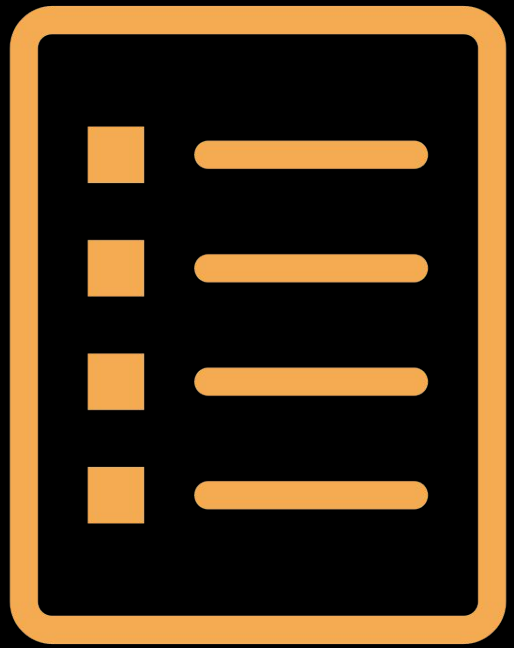
Components

Sketch the major system components

Data and request flow

Describe how the components interact

ToDo

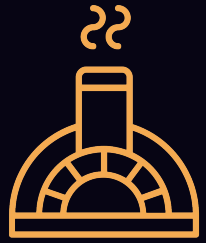


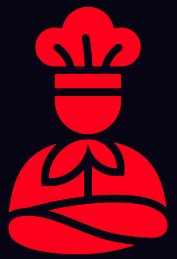
- Identify the entities
- Design the API
- Describe the data flow

Scaling Up

Scaling Up

how do we scale our system?





Scaling

more power

vertical scaling

more machines

horizontal scaling

Scaling

"scaling up"

Vertical Scaling

Definition	increase performance by adding more power (CPU, RAM, GPU, etc) to a machine
Pros	• simple to implement • no code changes required • easier maintenance
Cons	• physical limits of hardware • expensive to scale up/down • decreased resiliency

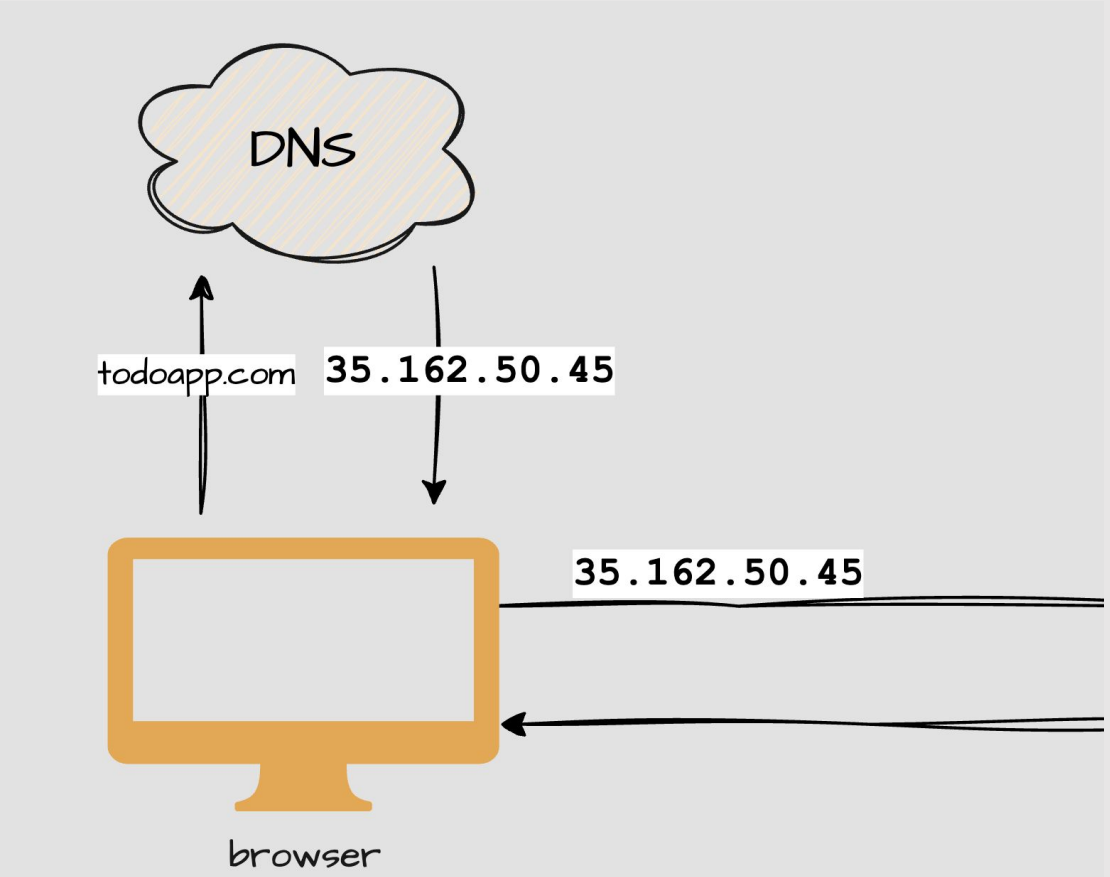
Scaling

"scaling out"

Horizontal Scaling

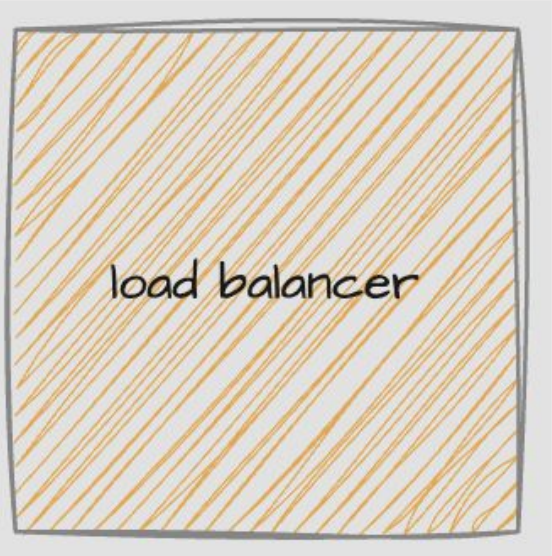
Definition	increase performance by adding more machines
Pros	• easily scales up/down with traffic • high availability • increased fault tolerance
Cons	• requires orchestration • can require code changes

Scaling



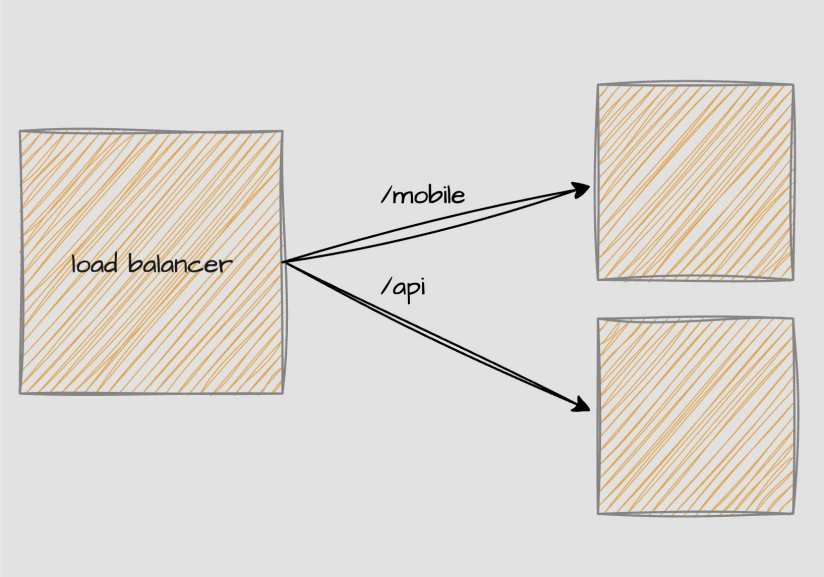
Domain Name System

Load Balancers



- distributes traffic evenly across services
- can act as a gateway for routing
- handles health checks and failovers

Load Balancers

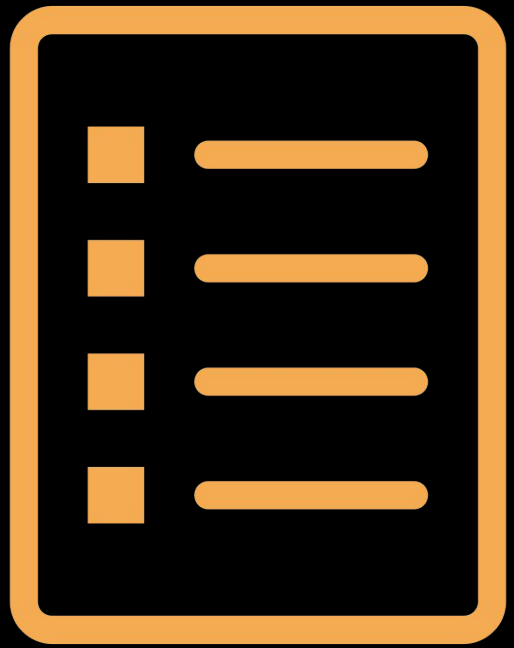


- distributes traffic evenly across services
- can act as a gateway for routing
- handles health checks and failovers

Load Balancers

Algorithm	Description
Round Robin	Requests distributed sequentially to each server in rotation
Least Connections	Sends requests to server with fewest active connections
Least Latency	Selects server with lowest response time
IP Hash	Uses client IP address to determine server (consistent hashing)

ToDo



- Scale our system!

Takeaways

High-level design

- High-level system design is about focusing on the main components and their interactions

- Scaling means trade-offs. Vertical scaling is easier but has limits. Horizontal scaling increases flexibility but requires more operational overhear.

Chapter 5
details

Data Storage

Data Storage

"At the end of the day, most of what you do is reading from and writing to a database."

Dimensions

structured vs unstructured

persistent vs
ephemeral
read-optimized vs write-optimized

consistency vs availability

Types of data

- images
- videos
-

Data Dtorage

Relational

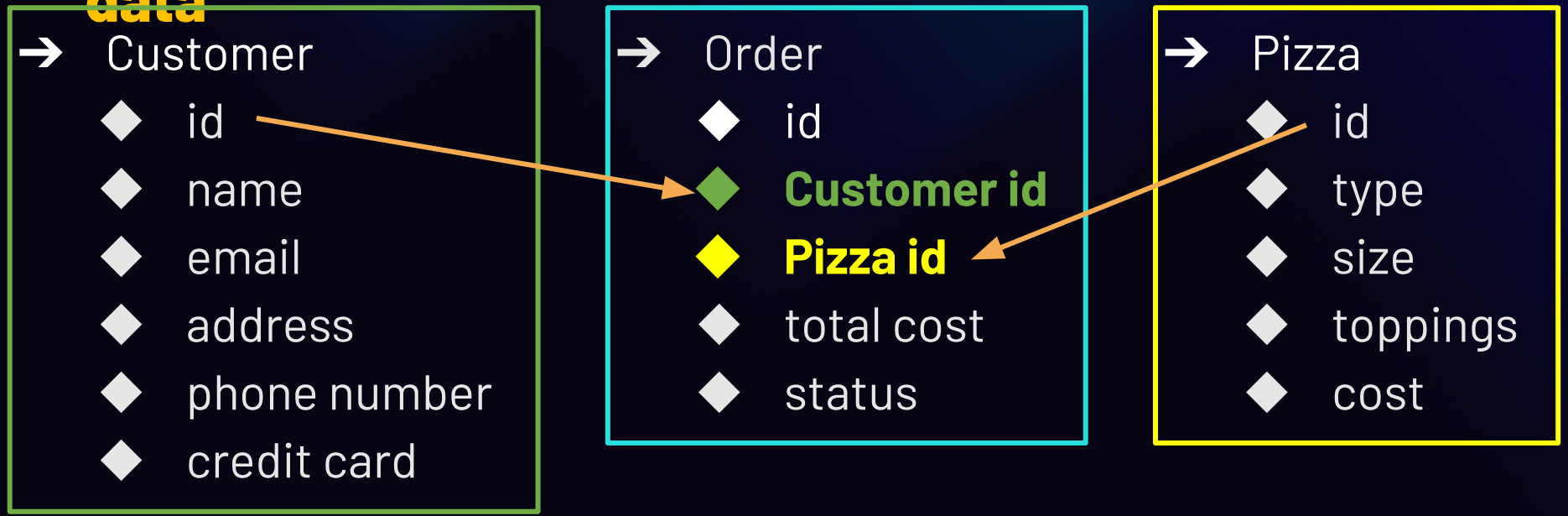
- “SQL”
- Structured data with relationships
- Enforced schema
- ACID transactions

Non-Relational

- “NoSQL”
- Semi or unstructured data
- Flexible

Relational

Structured data



Relational

Atomicity

Each transaction is all-or-nothing. Either every operation succeeds, or none do

Consistent

Transactions always follows the rules set for the database

Isolated

Concurrent transactions do not interfere with each other. Each runs as if it were alone

Durable

Once a transaction is saved, its changes are permanent

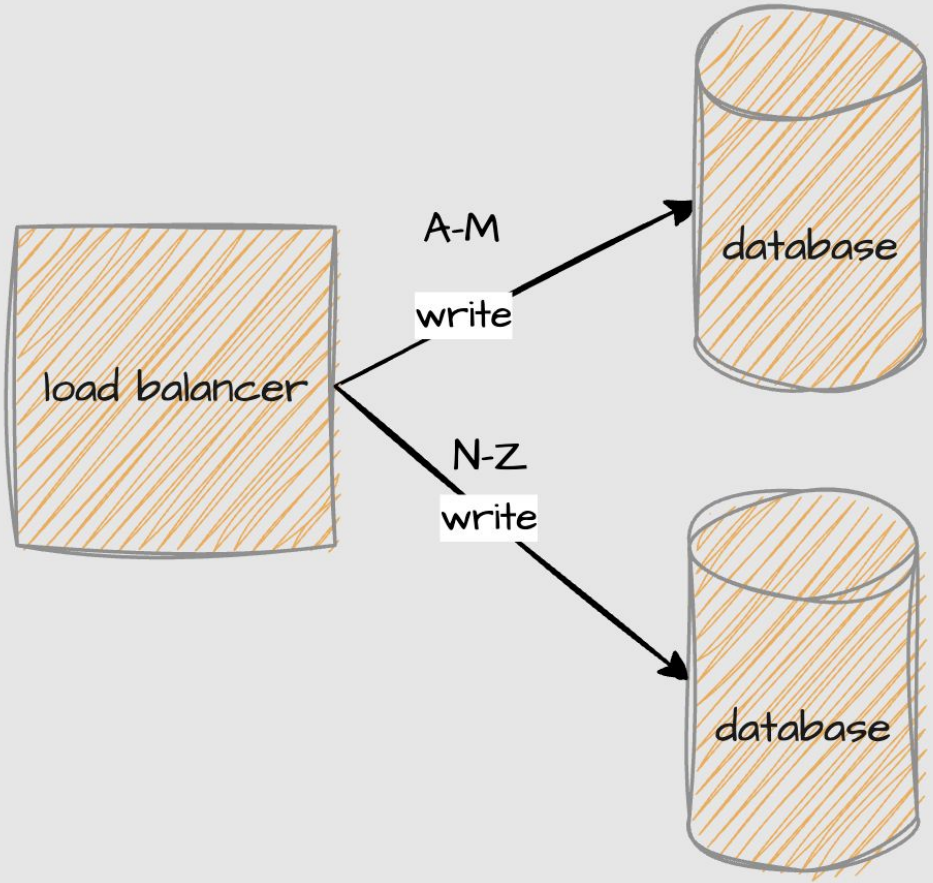
Non-relational

Document	Stores data as flexible, structured documents (like JSON)
Key-valu e	Stores data as simple pairs of a unique key and its value
Column	Organizes data into columns instead of rows for fast retrieval of similar data
Graph	Stores data as nodes and relationships, making it easy to represent and query connections

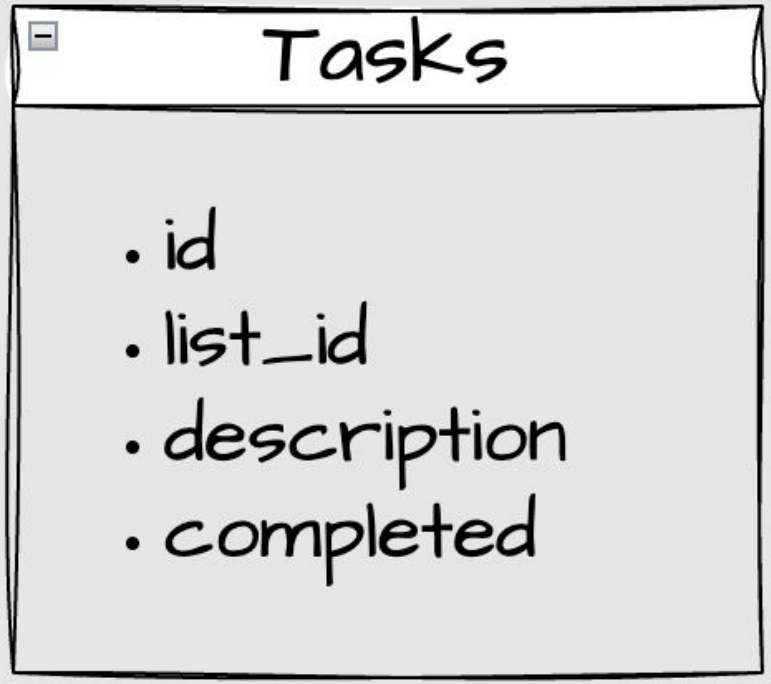
Data Storage

how do scale our data storage?

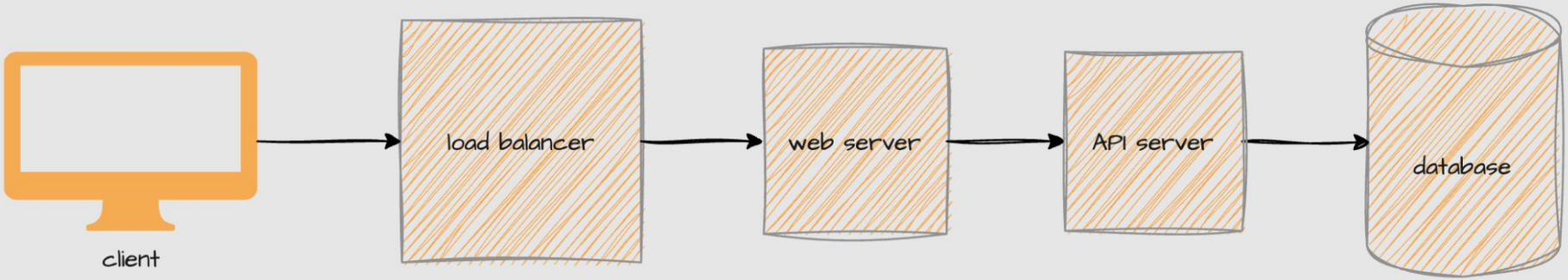
Sharding



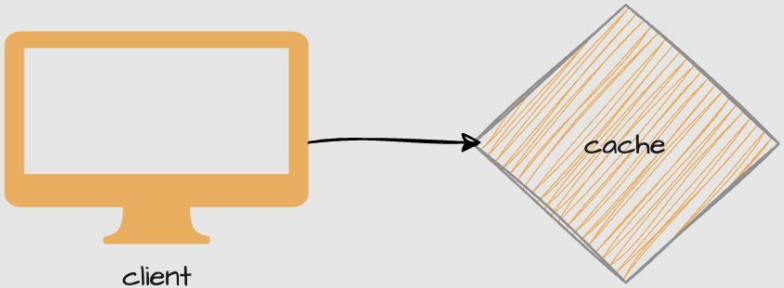
Partitioning



Caching



which is going to be faster?



Caching

- Reduces latency
- Improves user experience
- Decreases system load
- Lowers costs

Always
Be
Caching

examples

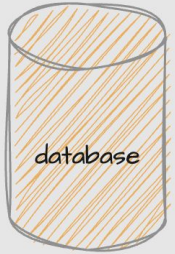
- local storage
- HTTP cache

types of data

- user preferences
- responses
- HTML, JS, CSS

- in-memory
- disc cache

- computed results
- API responses



- query cache
- buffer pool

- frequently accessed read data
- indexes
- query results

- Fastly, Akamai
- Open-connect

- static assets

Cache Invalidation

how do we keep our cache up to date?

Wrap