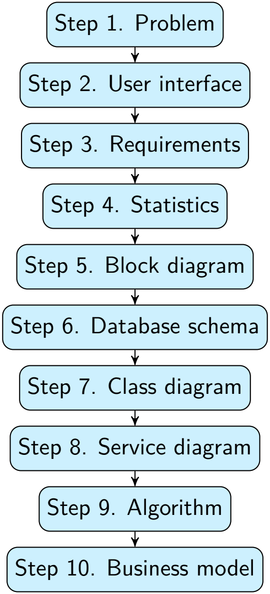
Contents

Contents

System design problem-solving template

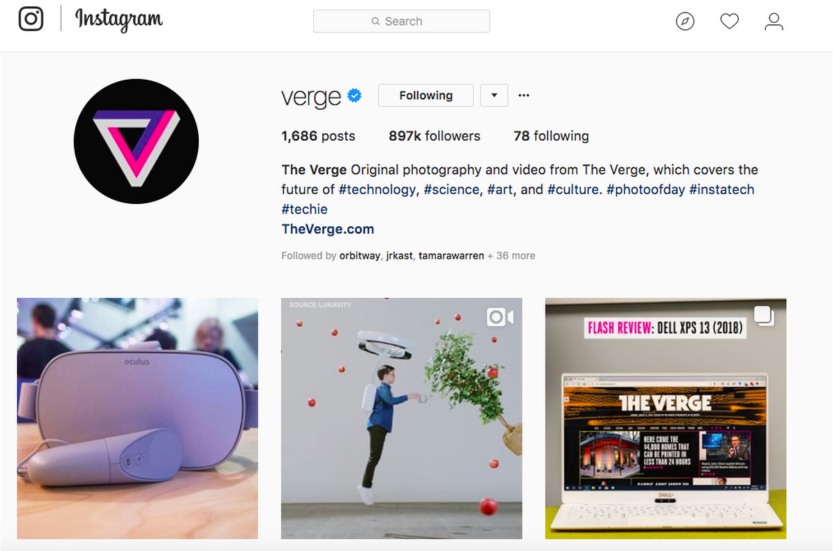


Instagram HOME

Step 1. Problem

- Design a simple social networking application like Instagram, where users can upload photos to share them with other users or view other user's photos.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Upload, download, view, like, and comment on photos
- Search photos/videos
- Follow users
- Generate and display news feed
- Support user login and authentication

Non-functional requirements

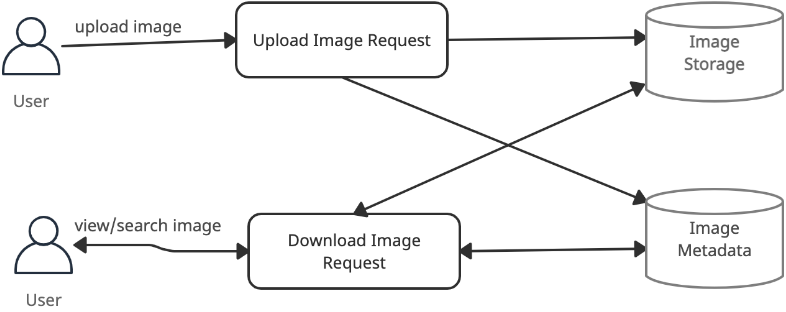
- High availability
- Low latency
- High reliability

Step 4. Statistics

Feature	Assumptions
Users	$500\mathcal{M}$ total, $1\mathcal{M}$ daily active
Photos	$2\mathcal{M}$/day (or 23/sec)
Photo size	200 KB
User activity	2 visits/day, 40 photos/visit
Storage	10 years

Parameter	Estimation
Photo space	$2\mathcal{M} \times 200$ KB $\times$ 365 days $\times$ 10 years $=$ 1425 TB
Daily upload volume	$2\mathcal{M} \times 200$ KB $=$ 400 GB
Upload bandwidth	400 GB / (24 $\times$ 3600 sec) $=$ 4.63 MB/sec
Daily download volume	$1\mathcal{M} \times 40 \times 2 \times 200$ KB $=$ 16000 GB
Download bandwidth	16000 GB / (3600 $\times$ 24 sec) $=$ 185 MB/sec

Step 5. Block diagram



Step 6. Database schema

Follow metadata	
UserID1,UserID2(PK)	int,int

User metadata	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
DateOfBirth	datetime
LastLogin	datetime
Creation date	datetime

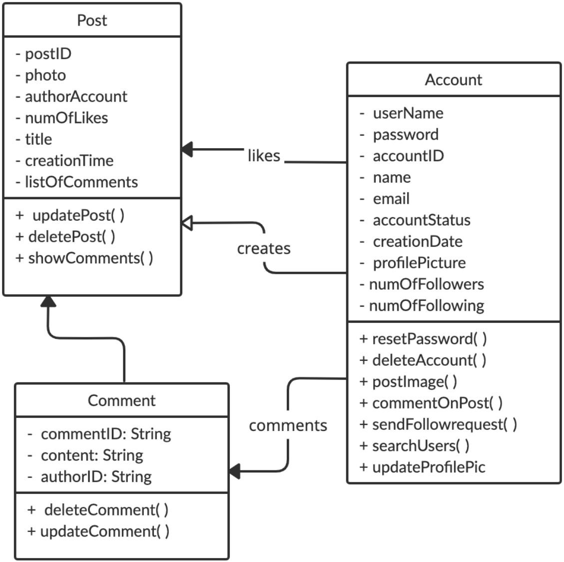
Photo metadata	
Photo ID(PK)	int
User ID	int
Photo path/Photo URL	varchar(256)
Photo latitude	int
Photo longitude	int
Photo caption	varchar(256)
Creation date	datetime

Step 6. Database schema

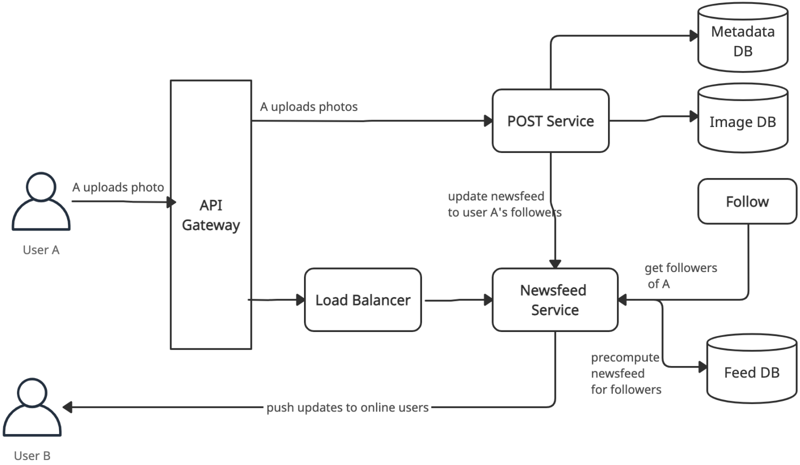
- Metadata

Table	Storage estimation
User	UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + DateOfBirth (4 bytes) + CreationDate (4 bytes) + LastLogin(4 bytes) = 68 bytes 500M (users) $\times$ 68 bytes = 32 GB
Photo	PhotoID (4 bytes) + UserID (4 bytes) + PhotoPath (256 bytes) + PhotoLatitude (4 bytes) + PhotoLongitude (4 bytes) + CreationDate (4 bytes) = 276 bytes $2\mathcal{M}$ $\times$ 276 bytes $\times$ 10 years = 1.88 TB
UserFollow	$500\mathcal{M}$ users $\times$ 500 followers $\times$ 8 bytes = 1.82 TB

Step 7. Class diagram



Step 8. Service diagram



Step 8. Algorithm

CREATETIMELINE($post, userId$) ▷ insert post in user's friends' newsfeed

Input: post, userId

1. Get each follower of a user
2. Append the post in the timeline hashmap for that follower

GETTIMELINE($user, time$)▷ fetch timeline for a user from a particular time

Input: user, time after which timeline should be generated

Output: Returns a list of posts to be shown to the user

1. get the minimum time in timeLine hash map beyond which we need to show the timeLine for the user into $boundTime$
2. iterate through the hash map from $boundTime$ to the end and append the post in a $posts$ list
3. **return** the $posts$ list to the user

Business Model

- Advertisements and sponsored posts
- Promotions

References

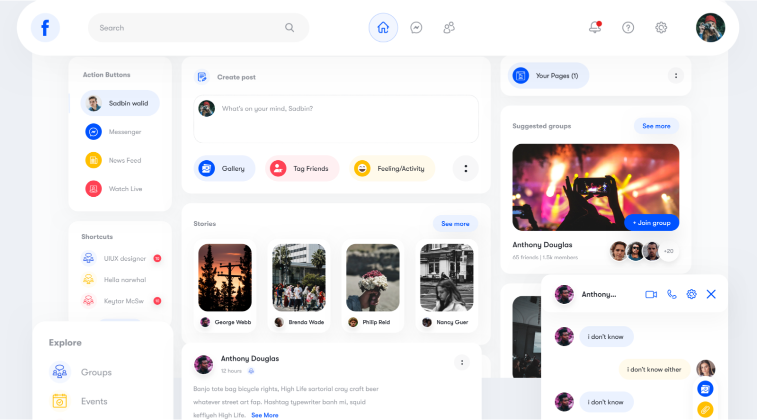
- Instagram Design in Grokking the System Design Interview
- System Design — Instagram by Dingding Wang
- System Design Mock Interview: Design Instagram Youtube video by Exponent
- Designing Instagram: System Design of News Feed Youtube Video by Gaurav Sen
- Instagram Engineering tech blog
- React-Instagram-Clone-2.0 Github repo:a basic instagram like application in React

Facebook HOME

Step 1. Problem

- Design a simple online social networking application like Facebook, where users can connect with other users to post and read messages.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Add/update profile and search/connect with friends
- Create posts, add pictures, and share videos
- Search/create/subscribe groups
- Generate newsfeed from friends' posts and updates from groups

Non-functional requirements

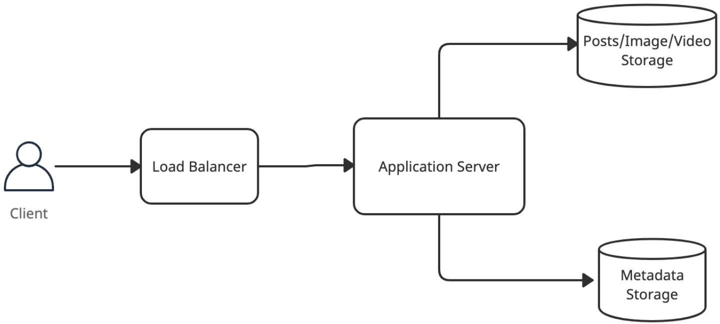
- High availability
- Low latency
- High reliability

Step 4. Statistics

Feature	Assumptions
Users	$1\mathcal{B}$ total, $200\mathcal{M}$ daily active
Posts	$100\mathcal{M}$/day (or 1157/sec) of 800 bytes each
Multimedia frequency	Every 5th post has a photo and 10th has a video
Average multimedia size	200 KB for photos and 2 MB for videos
Average number of friends	200
User activity	2 visits/day, 50 posts/visit (or $20\mathcal{B}$/day)
Storage	10 years

Parameter	Estimation
Posts space	$100\mathcal{M}$ × 800 bytes × 365 days × 10 years = 292 TB
Photo space	$20\mathcal{M}$ × 200 KB × 365 days × 10 years = 14600 TB
Video space	$10\mathcal{M}$ × 2 MB × 365 days × 10 years = 73000 TB
Daily upload volume	80 GB + 4000 GB + 20000 GB = 24080 GB
Upload bandwidth	24080 GB / 3600 * 24 sec (or 278 MB/sec)
Download volume	$20\mathcal{B}$×800 bytes + $4\mathcal{B}$×200 KB + 2B×2 MB = 4816 TB
Download bandwidth	4816 TB / 3600 * 24 sec (or 56 GB/sec)

Step 5. Block diagram



Step 6. Database schema

User metadata	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
Password	varchar(32)
DateOfBirth	datetime
LastLogin	datetime
Creation date	datetime
Phone	varchar(15)
Gender	varchar(2)
Work	varchar(64)
Education	varchar(64)

Post metadata	
Post ID (PK)	varchar(64)
Author ID	varchar(20)
Post path	varchar(256)
Creation Date	datetime
Likes	int
Media ID	varchar(64)

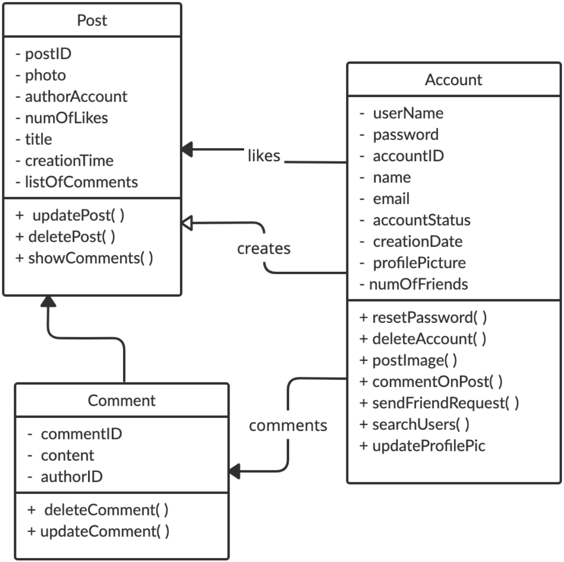
Media metadata	
Media ID(PK)	varchar
Type	int
Media path/URL	varchar(256)
Creation date	datetime

Step 5. Database schema

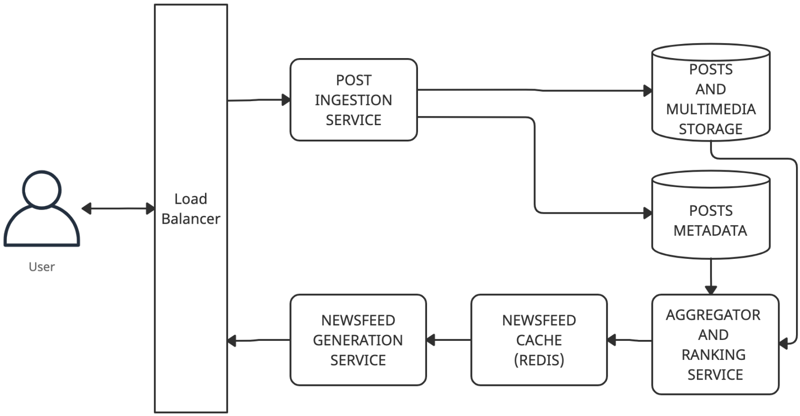
- Metadata

Table	Storage estimation
User	UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + DateOfBirth (4 bytes) + CreationDate (4 bytes) + LastLogin(4 bytes) + Phone(15 bytes) + Gender(2 bytes) + Work(64 bytes) + Education(64 bytes) = 213 bytes $500\mathcal{M}$(users) * 213 bytes = 100 GB
Photo	PhotoID (4 bytes) + UserID (4 bytes) + PhotoPath (256bytes) + PhotoLatitude (4 bytes) + PhotoLongitude(4 bytes) + CreationDate (4 bytes) + Likes (4 bytes) + Media ID (64 bytes) = 344 bytes $2\mathcal{M}$ * 344 * 10 years = 2.34 TB height

Step 6. Class diagram



Step 7. Service Diagram



Step 8. Algorithm

$\textsc{createTimeLine}(post, userId)$ ▷ insert post in user's friends' newsfeed

Input: post, userId
1. Get each friend of a user
2. Append the post in the timeline hashmap for that friend

$\textsc{getTimeLine}(user, time)$ ▷ fetch timeline for a user from a particular time

Input: user, time after which timeline should be generated
Output: Returns a list of posts to be shown to the user
1. get the minimum time in timeLine hash map beyond which we need to show the timeLine for the user into $boundTime$
2. iterate through the hash map from $boundTime$ to the end and append the post in a $posts$ list
3. **return** the $posts$ list to the user

Business Model

- Advertisements and sponsored posts
- Promotions

References

- Facebook Newsfeed Design in Grokking the System Design Interview
- Facebook System Design youtube video by Codekarle
- Design Facebook Messenger Youtube Video by Exponent
- Facebook Engineering tech blog

Search Engine HOME

Step 1. Problem

- Design a simple search engine to store, organize billions of web pages available on the internet, find and present the most relevant suggestions to the queries that the users type into the search bar

Step 2. User interface

Step 3. Requirements

Functional requirements

- Find pages from a search box
- Rank search results based on relevance
- Provide a short summary with links to actual pages

Non-functional requirements

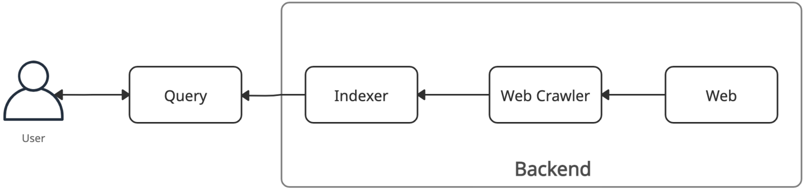
- High scalability
- Low latency
- High availability
- High extensibility

Step 4. Statistics

Feature	Assumptions
Websites to crawl	$1\mathcal{B}$
# web pages in a website	15
Average page size with text	100 KB
Page metadata	500 bytes
# english words and nouns to be indexed	300000 words + 200000 nouns = 500000 words
Indexed word length	5 characters (5 bytes)
Document ID.	64 bytes
# indexed words in each doc	400

Parameter	Estimation
# pages to be crawled	$1\mathcal{B} \times 15 = 15\mathcal{B}$ pages
Document storage	$15\mathcal{B} \times$ (100 KB + 500 bytes) = 1.5 PB
Document ID storage	$15\mathcal{B} \times$ 64 bytes = 960 GB
Key storage in index	500000 words $\times$ 5 bytes = 2.5 MB
Value/DocIDs storage in index	90 GB $\times$ 400 words = 36 TB

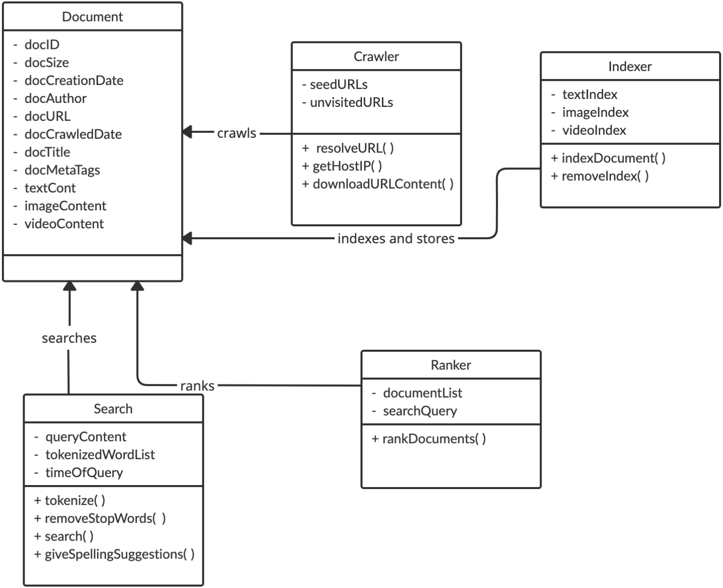
Step 5. Block diagram



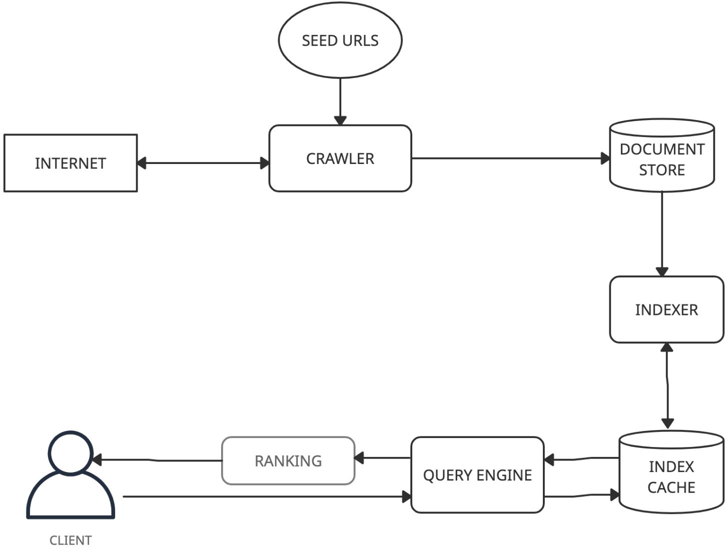
Step 6. Database schema

Document	
Doc ID (PK)	varchar(30)
Doc size	int
Creation date	datetime
Author	varchar(60)
DocURL	varchar(1000)
Doc title	varchar(200)
Doc content	varchar(2000)
Meta tags	varchar(300)

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

Crawl(seed URLs list) ▷ Crawl billions of documents on the internet

Input: seed URLs list

1. Get initial seed URLs and store them in the unvisited URLs list
2. Pick a URL from the unvisited URL list
3. Determine the IP address of its host-name
4. Establish a connection with the host to download the document
5. Parse the document contents to look for new URLs
6. Add the new URLs to the list of unvisited URLs
7. Parse and store/index the document
8. Go back to step 2

Index(*document*) ▷ Index the crawled document to search it later

Input: document to be indexed

1. For every crawled document, tokenize the document
2. Remove stop words and retrieve the key words
3. Iterate through the key words
4. For every key word, store the current document ID against that key word in the index

Step 10. Business model

- Advertisements

References

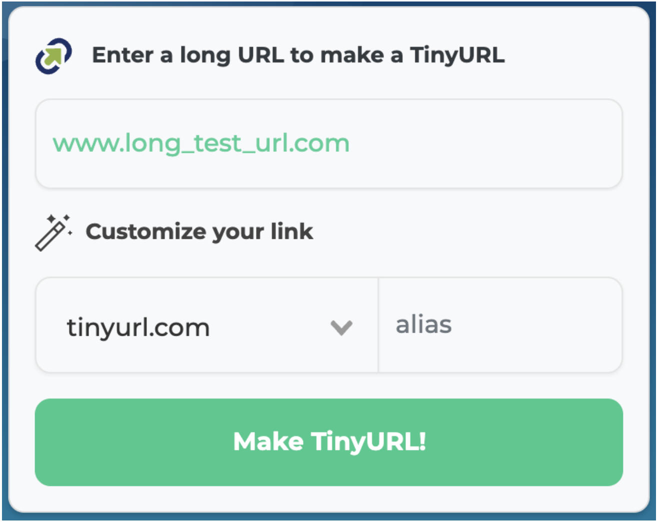
- Design a search engine medium article
- Design a mini google search medium article
- Google search architecture and components
- Google search indexing and searching youtube video
- Google search engine system design
- Google search engine high level system design
- Google search engine youtube video

TinyURL HOME

Step 1. Problem

- Design a URL shortening service like TinyURL. This will provide a short aliases redirecting to long URLs.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Generate short, unique aliases of long URLs
- Redirection to long URL on accessing short alias
- Customize short link
- Link expiry according to time specified by user

Non-functional requirements

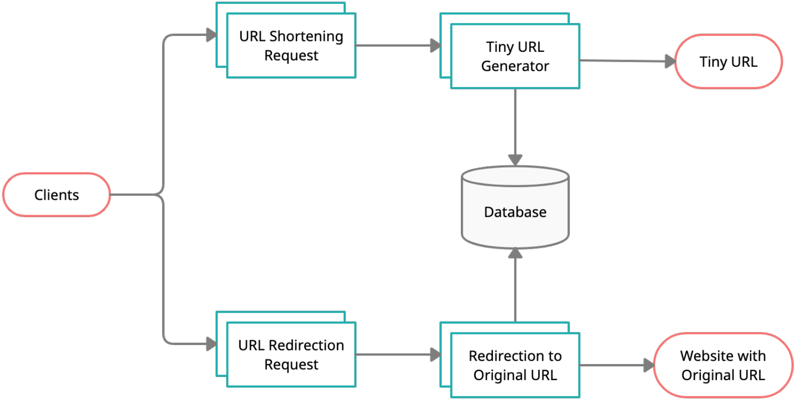
- High availability
- Low latency
- Short links should be unpredictable

Step 4. Statistics

Feature	Assumptions
Shortenings	500 million requests per month = 200 URL/sec
Redirections	50 billion requests per month = 20000 redirections/sec
Storage time	5 years
Total URLs	500 million $\times$ 5 years $\times$ 12 months = 30 billion URLs

Parameter	Estimation
URL storage space	30 billion $\times$ 500 bytes = 15 TB
Bandwidth	20K $\times$ 500 bytes = 10 MB/s
Redirection requests per day	20K $\times$ 3600 seconds $\times$ 24 hours = 1.7 billion
Cache memory	0.2 $\times$ 1.7 billion $\times$ 500 B = 170 GB

Step 5. Block diagram

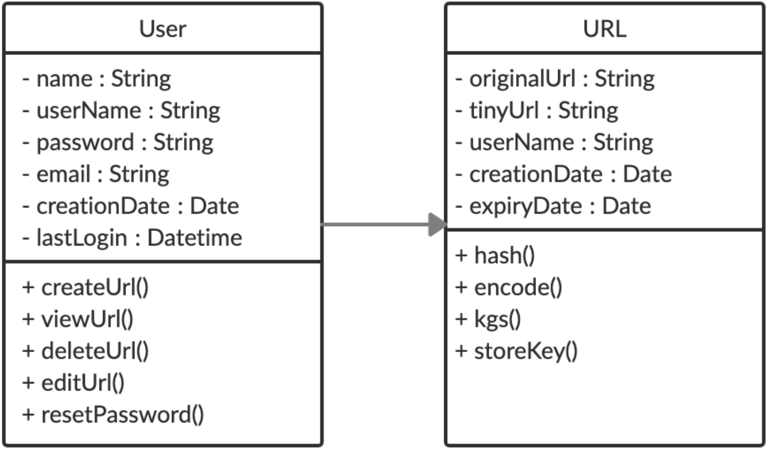


Step 6. Database schema

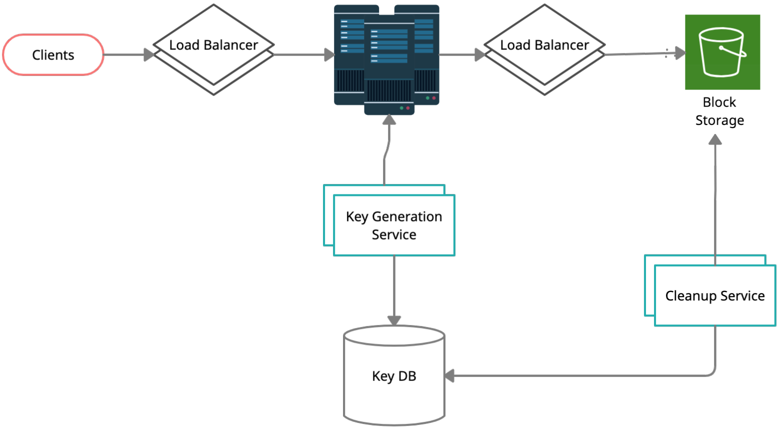
URL	
Hash (PK)	varchar(16)
Original URL	varchar(512)
Creation Date	datetime
Expiration Date	datetime
User ID	int

User	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
Creation Date	datetime
Last Login	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

TinyURLGenerator($url, user$) ▷ Encode actual URL

Input: URL, user ID
Output: Generate a tiny URL for the given URL
(#Unique tiny URLs with 6 letters, 64-bit encoding $= 6^{64} = 68.7$ billion)
1. Hash (URL + user ID) using a hashing algorithm, e.g. SHA256/MD5
2. Compress the generated hash to 6 letters using a compression algorithm
3. Return the tiny url consisting of 6 letters

TinyURLGenerator($url, user$) ▷ Key Generation Service

Input: URL, user ID
Output: Generate a tiny URL for the given URL
(Key DB size - 6 characters × 68.7 billion unique keys = 412 GB)
1. Fetch available key (tiny url) from Key table and assign to URL
2. Move unique key to Used-Key table
3. Return the tiny url consisting of 6 letters

Step 10. Business Model

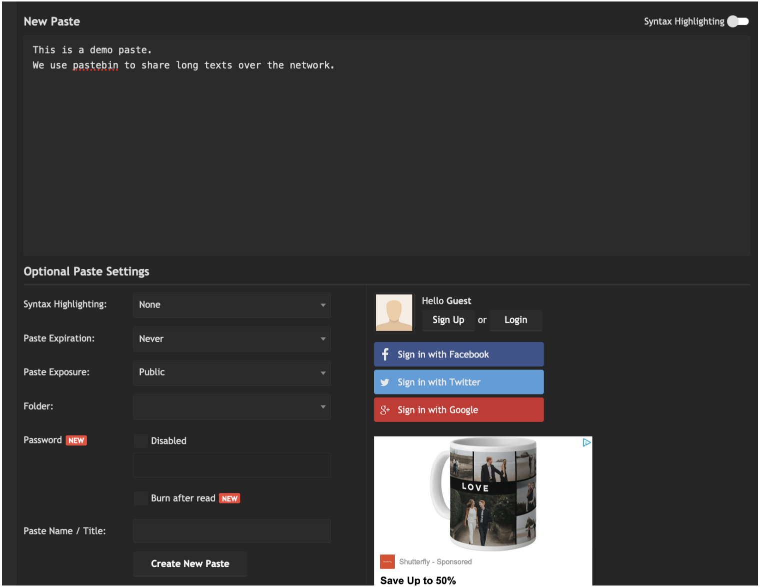
- Advertisements
- Freemium model

Paste bin HOME

Step 1. Problem

- Design a simple Pastebin where users can store plain text over the internet and generate unique URLs to access that data.

Step 2. User Interface



Step 3. Requirements

Functional requirements

- Upload/paste text data
- Get unique URL for accessing the uploaded data
- Data and links expiry after a specified time
- Custom alias chosen by the user

Non-functional requirements

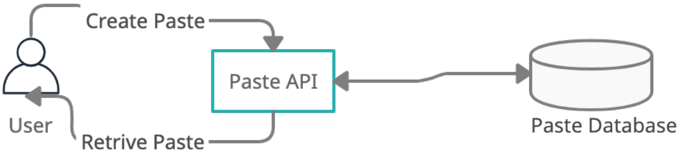
- High reliability
- Low latency
- High availability
- Links should not be guessable

Step 4. Statistics

Feature	Assumptions
New pastes	1 $\mathcal{M}$ per day (or 12/sec)
Paste reads	5 $\mathcal{M}$ per day (or 58/sec)
Paste size	10 KB average
Storage time	10 years

Parameter	Estimation
Data space	1 $\mathcal{M}$ $\times$ 10 KB $\times$ 365 days $\times$ 10 years $=$ 36.5 TB
Unique strings	base64 encoding- 6 letter strings ($64^6 = 68.7B$)
Key storage	3.6 TB $\times$ 10 years $\times$ 6 $=$ 22 GB
Daily upload volume	1 $\mathcal{M}$ $\times$ 10 KB $=$ 10 GB
Upload bandwidth	10 GB / 3600 $\times$ 24 secs $=$ 115.7 KB/sec
Daily download volume	5 $\mathcal{M}$ $\times$ 10 KB $=$ 50 GB
Download bandwidth	50 GB / 3600 $\times$ 24 secs $=$ 0.58 MB/sec

Step 5. Block diagram



Step 6. Database schema

Paste	
urlHash (PK)	varchar(16)
contentKey	varchar(512)
expirationDate	datetime
userID	int
creationDate	datetime

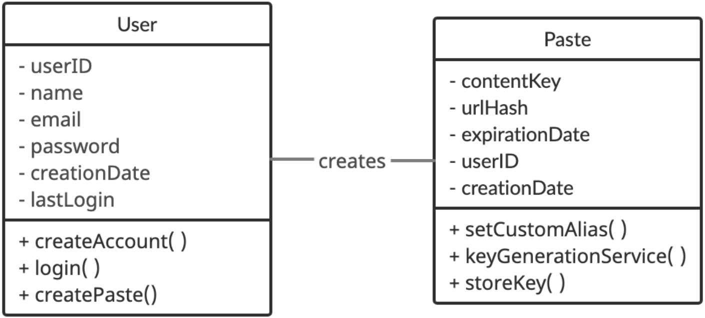
User	
userID (PK)	int
name	varchar(20)
email	varchar(32)
password	int
creationDate	datetime
lastLogin	datetime

Step 6. Database schema

- Metadata

Table	Storage estimation
User	userID (4 bytes) + name (20 bytes) + email (32 bytes) + password (8 bytes) + creationDate (4 bytes) + lastLogin (4 bytes) = 68 bytes 5 $\mathcal{M}$ (users) $\times$ 68 bytes = 0.32 GB
Paste	urlHash (16 bytes) + userID (4 bytes) + contentKey (512 bytes) + expirationDate (4 bytes) + creationDate (4 bytes) = 540 bytes 1 $\mathcal{M}$ $\times$ 540 $\times$ 10 years = 1.9 TB

Step 7. Class diagram



Step 8. Algorithm - Key Generation

RANDOMKEYGENERATOR($paste, user$) ▷ Generate key for a paste

Input: paste, userID
Output: Generate a key for the given paste
(#Possible unique keys with base64 encoding $= 6^{64} = 68.7\ \mathcal{B}$)

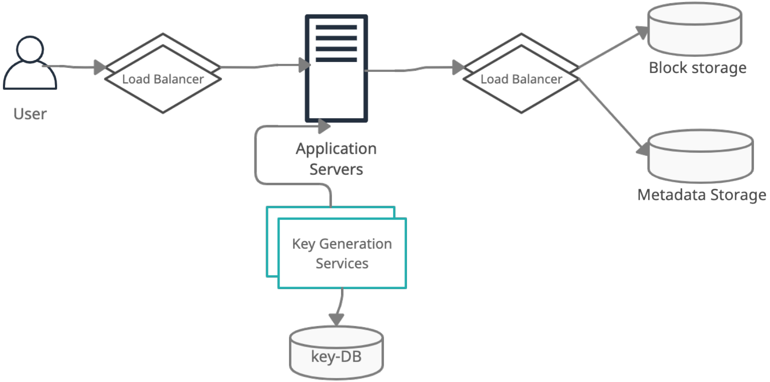
1. Generate a 6-letter random string upon receiving write request
2. Store the contents of paste and generated key in database if no collision
3. Regenerate the random string if collision occurs
4. Return error if user's custom key is already in the database

KEYGENERATIONSERVICE($paste, user$) ▷ Generate key for a Paste

Input: paste, userID
Output: Generate a key for the given Paste
(#Possible unique 6-letter keys with base64 encoding $= 6^{64} = 68.7\ \mathcal{B}$)
(These unique keys are generated beforehand and stored on key-DB)

1. Provide a key from the key-DB for the given paste
2. Move the provided key to used key database
3. Return the provided key

Step 8. Service diagram



Step 9. Business Model

- Revenue generated from donations and advertisements
- Freemium model

References

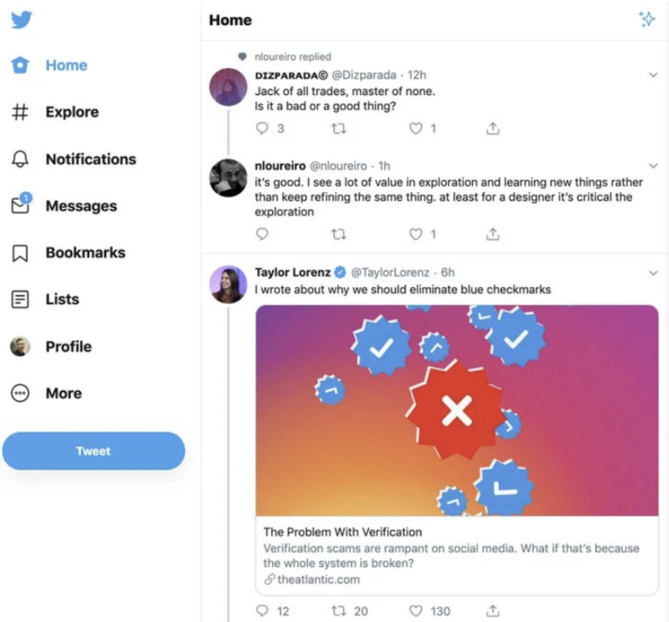
- Paste bin Design in Grokking the System Design Interview
- Designing Paste bin by CrackFAANG
- System Design Primer- Paste bin

Twitter HOME

Step 1. Problem

- Design an online social networking service where users post and read short 140-character messages called "tweets."

Step 2. User interface



Step 3. Requirements

Functional requirements

- Post new tweets (includes photos and videos)
- Follow other users and/or topics
- Mark tweets as favorites
- Display user timeline with top tweets

Non-functional requirements

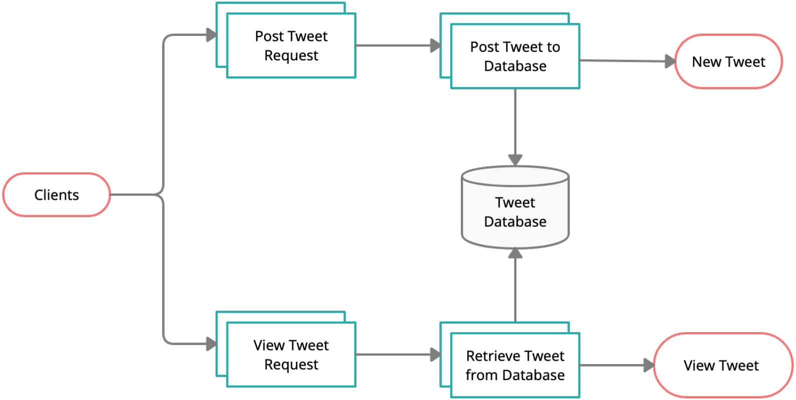
- High availability (over consistency)
- Acceptable latency of 200 ms

Step 4. Statistics

Feature	Assumptions
Favourites	200 million Users × 5 favorites = 1 billion favorites/day
Tweet views	200 million Users × ((2 + 5) × 20 tweets) = 28 billion/day

Parameter	Estimation
Tweet space	100 million × (280 + 30) B = 30 GB/day
Photo space	20 million × 200KB = 4 TB/day
Video space	10 million × 2 MB = 20 TB/day
Incoming bandwidth	24 TB/day = 290 MB/s
Tweet view bandwidth	28 billion × 280 B / 86400 = 93MB/s
Photo view bandwidth	28 billion/5 × 280 B / 86400 = 13 GB/s
Video view bandwidth	28 billion/10/3 × 280 B / 86400 = 22 GB/s
Outgoing bandwidth	35 GB/s

Step 5. Block diagram



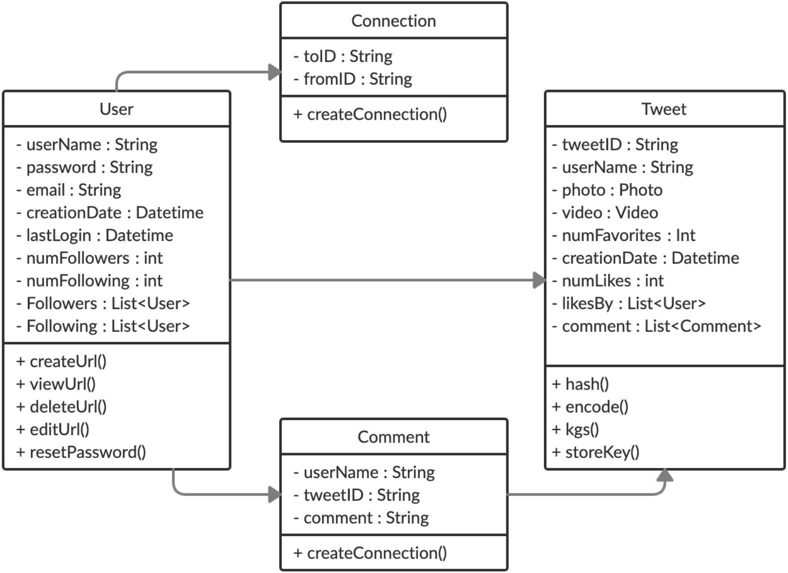
Step 6. Database schema

Tweet	
Tweet ID (PK)	int
User ID	int
Content	varchar(140)
Tweet latitude	int
Tweet longitude	int
User latitude	int
User longitude	int
Creation date	datetime
Num favorites	int
Top favorites	varchar(1000)

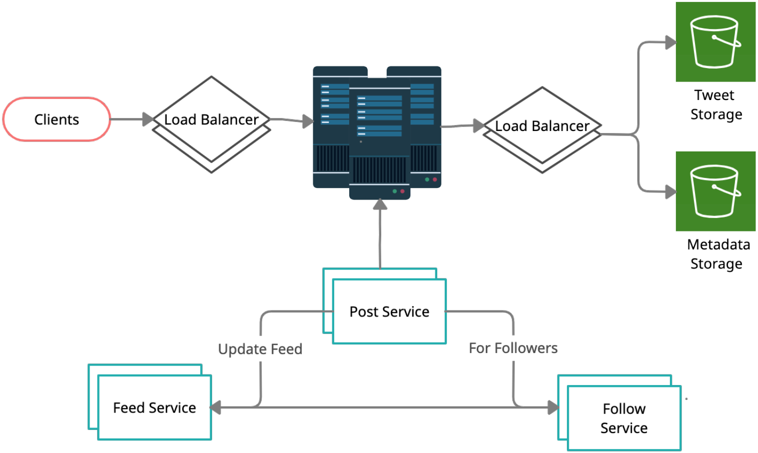
User	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
Date of birth	datetime
Creation date	datetime
Last login	datetime

User follow	
User ID1 (PK)	int
User ID2	int

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

PostTweet(user,tweet)

Input: User and a tweet
Output: Post a new tweet and update feed of user's followers

1. **for** each follower of the user **do**
2. add tweet to the follower's feed
3. add tweet to user's tweet list

GetFeed(user,time)

Input: User and time
Output: List of tweets to be shown to the user posted after the given time

1. **for** each tweet in user's feed **do**
2. output tweet posted after given time

Step 10. Business model

- Advertisements
- Promotions
- Data licensing

Whatsapp HOME

Step 1. Problem

- WhatsApp is an instant messaging service that supports one-on-one and group chats between users through mobile and web interfaces.

Step 2. User interface

Step 3. Requirements

Functional requirements

- One-on-one and group conversations
- Track of online/offline status
- Send text, pictures, audios, videos and other files
- Sent/delivered/read notifications
- Audio/video calls

Non-functional requirements

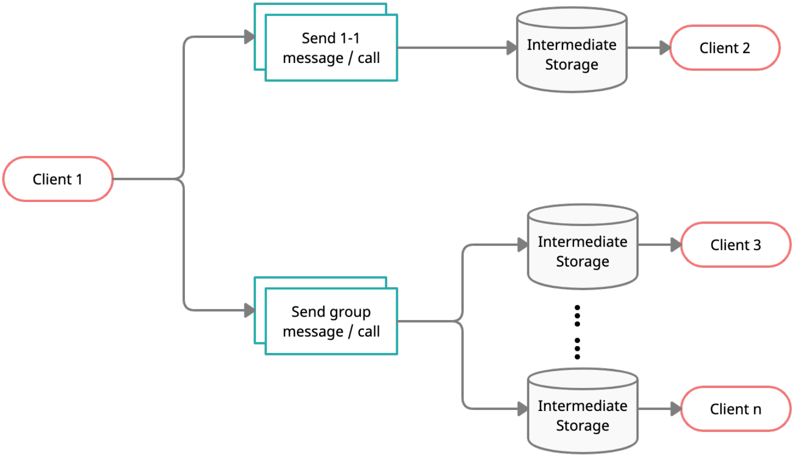
- Minimum latency
- High availability
- High consistency
- Persistent storage

Step 4. Statistics

Feature	Assumptions
Daily users	250 million users
Daily messages	40 messages/user = 10 billion messages/day

Parameter	Estimation
Message space	10 billion messages × 140 bytes = 1.4 TB/day
Photo space	1 billion photos × 200 KB bytes = 200 TB/day
Video space	500 million videos × 2 MB = 1000 TB/day
File space	500 million files × 140 bytes = 0.1 TB/day
Total storage	1200 TB × 365 days × 10 years = 4285 PB
Bandwidth	1200 TB/day = 15 GB/s
Voice call bandwidth	16 KB/second for 4G
Video call bandwidth	80 KB/second for 4G

Step 5. Block diagram

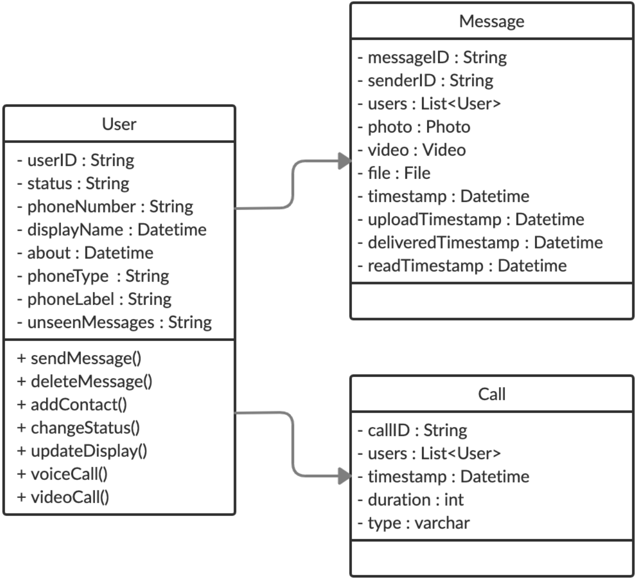


Step 6. Database schema

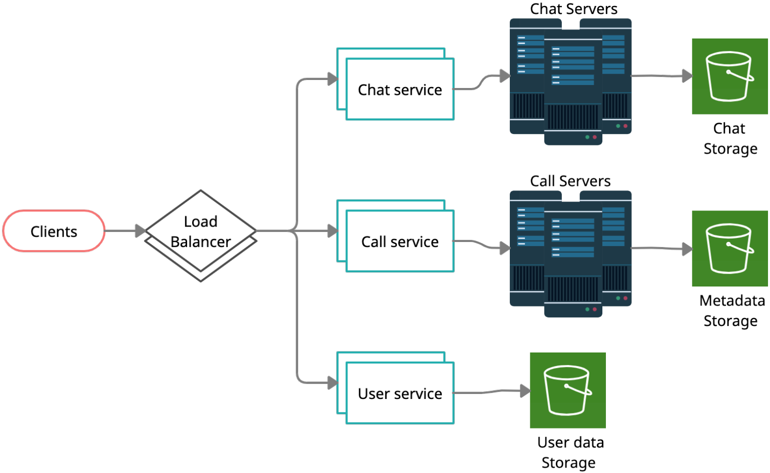
User	
User ID (PK)	int
Status	varchar(140)
Number	int
Display name	varchar(140)
About	varchar(140)
Phone type	varchar(140)
Phone label	varchar(140)
Unseen messages	int
Profile photo	blob

Message	
Message ID (PK)	int
Sender ID	int
Group ID	int
Timestamp	datetime
Media	blob
Upload timestamp	datetime
Delivered timestamp	datetime
Seen timestamp	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

EncryptMessage(sender,message,recipient)

Input: User,message and a recipient
Output: Encrypt message sent from user to recipient

1. Sender encrypts message using public key from server
2. Encrypted message is sent to recipient
3. Recipient decrypts message using private key and public key from server

MessageDelivery(sender,message,recipient)

Input: User,message and a recipient
Output: Send message from user to recipient

1. User sends a message to recipient
2. If user is online, message is sent to server. Message status is sent
3. If recipient is online, send from server. Message status is delivered.
4. If recipient reads message, message status as read.

Step 10. Business model

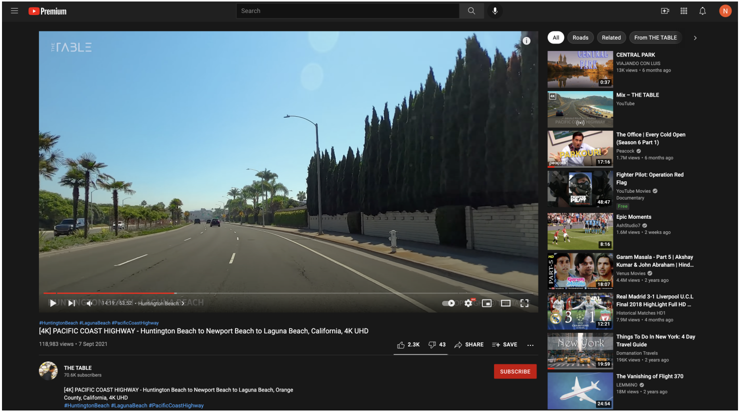
- Business API
- P2P payments

Youtube HOME

Step 1. Problem

- Youtube is one of the most popular video sharing websites in the world. Users of the service can upload, view, share, rate, and report videos as well as add comments on videos.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Upload/view/share/like/dislike videos
- Subscribe to channels
- Recommend videos
- Record statistics of videos

Non-functional requirements

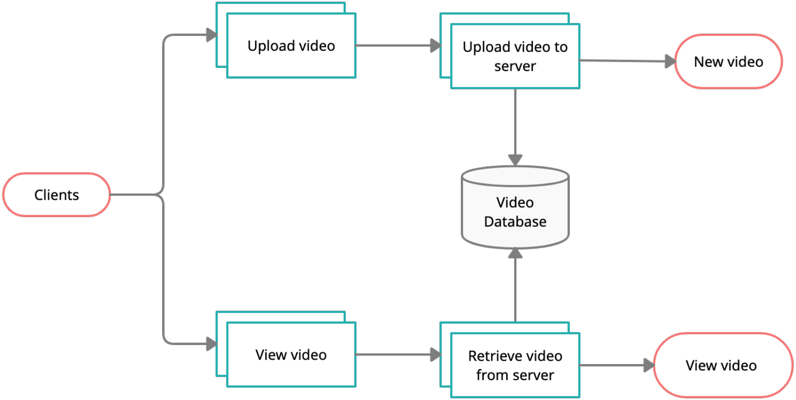
- High reliability
- High availability
- Minimum latency

Step 4. Statistics

Feature	Assumptions
Daily users	800 million users
Daily views	5 views/user = 4 billion views
Average video size	50 MB
Upload view ratio	1:200
Videos uploaded	200 uploads/second
Video length uploaded	500 hours/minute

Parameter	Estimation
Storage	500 hours × 60 minutes × 50 MB = 25 GB/second
Upload bandwidth	500 hours × 60 minutes × 10 MB = 5 GB/second
Download bandwidth	5 GB/second × 200 = 1 TB/second

Step 5. Block diagram

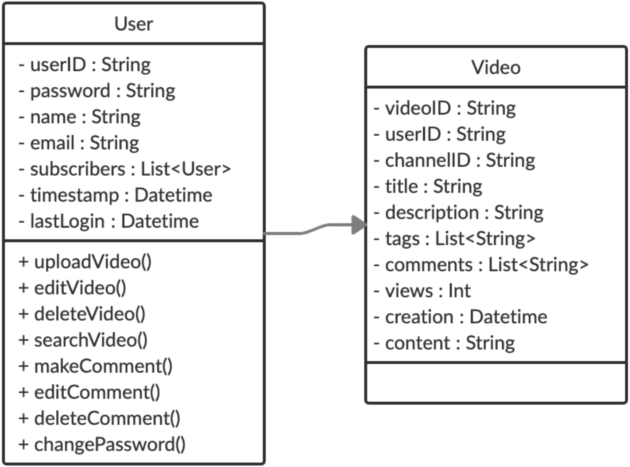


Step 6. Database schema

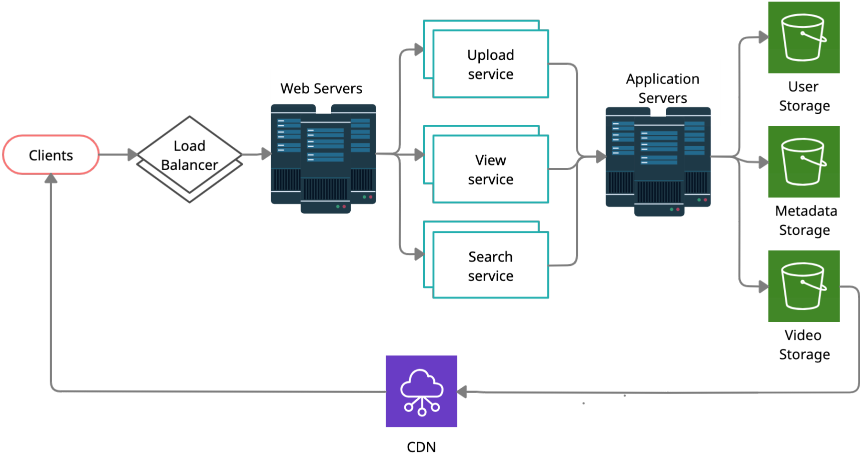
User	
User ID (PK)	int
Name	varchar(30)
Email	varchar(32)
Subscribers	int
Creation timestamp	datetime
Last login	datetime

Video	
Video ID (PK)	int
User ID	int
Channel ID	int
Likes	int
Dislikes	int
Title	varchar(140)
Description	varchar(300)
Views	int
Creation timestamp	datetime
Video	blob

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

Recommend(user)

Input: User's watch history, subscriptions, likes and dislikes
Output: Recommend videos personalised for the user

1. Rank videos based on the following factors
2. Personalised - User's topic interests, watch history, channels
3. Performance - Views, average view duration, likes, dislikes, surveys
4. External - Trending topics, seasonality and competition
5. User receives top ranked videos as recommendation

Step 10. Business model

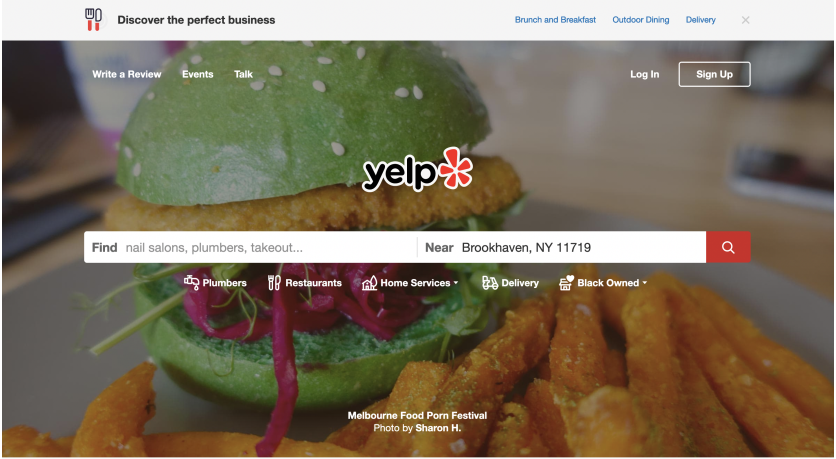
- Premium Subscriptions
- Advertisements
- Channel memberships
- Super chats, super stickers and merchandise

Yelp HOME

Step 1. Problem

- Design a simple Yelp like service where users can search for nearby places like restaurants, theatres, etc. for user generated ratings, reviews and other details.

Step 2. User Interface



Step 3. Requirements

Functional requirements

- Add images/text/ratings as a review for a particular place
- Get all nearby places based on user's current location
- Add, update or delete places

Non-functional requirements

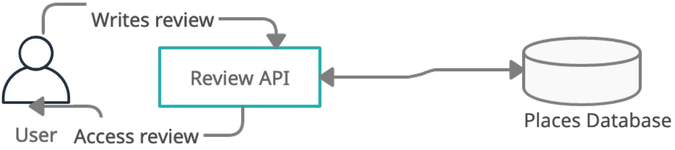
- Heavy search load
- Low latency

Step 4. Statistics

Feature	Assumptions
Places	500 $\mathcal{M}$
Queries	100k /sec
Photos	200 KB
Review size	512 bytes average
Places growth	20% /year

Parameter	Estimation(Assuming Quadtree)
Tree storage	500 $\mathcal{M} \times 8 \times 3$ bytes $=$ 12 GB
Photo Space	500 $\mathcal{M} \times 10 \times 250$ KB $=$ 1250 TB
Daily upload volume	0.5 $\mathcal{M}$ (reviews/day)$\times$ 512 bytes $\times$ 500 (photos) $\times$ 200 KB= 25.6 TB
Upload bandwidth	25.6 TB / 3600 $\times$ 24 seconds $=$ 29 GB/sec
Daily download volume	2 $\mathcal{M} \times$ 2 MB $=$ 4 TB
Download bandwidth	4 TB / 3600 $\times$ 24 seconds $=$ 46.3 MB/sec

Step 5. Block diagram



Step 6. Database schema

Places	
locationID (PK)	int
name	varchar(256)
latitude	decimal
longitude	decimal
description	varchar(512)
category	char

Photos	
locationID (PK)	int
photoURL	varchar(256)

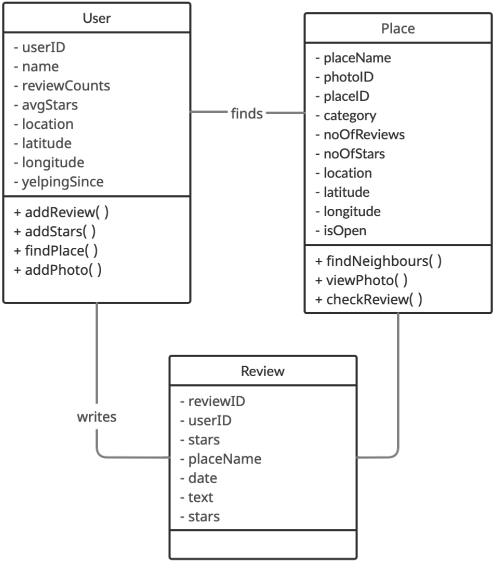
Reviews	
locationID (PK)	int
reviewID	int
reviewText	varchar(512)
rating	int

Step 6. Database schema

- Metadata

Table	Storage estimation
Places	locationID (8 bytes) + name (256 bytes) + latitude (8 bytes) + longitude (8 bytes) + description (512 bytes) + category (1 bytes) = 793 bytes 500 $\mathcal{M}$ (places) $\times$ 793 bytes = 0.4 TB
Reviews	locationID (8 bytes) + reviewID (4 bytes) + reviewText (512 bytes) + rating (1 byte) = 525 bytes 500 $\mathcal{M}$ (places) $\times$ 525 bytes = 0.26 TB
Photos	locationID (8 bytes) + photoURL (256 bytes) = 264 bytes 500 $\mathcal{M}$ (places) $\times$ 264 bytes = 0.13 TB

Step 7. Class diagram



Step 8. Algorithm

QuadTreeGenerator($Places$) ▷ Generate a quad tree for locations

Input: locationID, latitude, longitude
Output: Generate a tree for all the locations
(#Locations to start with - 500 $\mathcal{M}$)

1. Start with one node as a grid for entire world
2. Break down that node into four grids based on locations
3. Repeat for each child node until no nodes are left with $>$500 locations

GridNeighborFinder($Places$) ▷ Find the neighboring grid

Input: locationID, latitude, longitude
Output: Find the neighboring grid for given location

1. Start from root node the contains user location and traverse down
2. Connect all the leaf nodes with a doubly linked list
3. Iterate the doubly linked list forward and backward
4. Return as you find the neighboring locations or exhaust the search

Step 8. Algorithm

TreePartitioning($Places$) ▷ Partition tree based on Locations

Input: locationID, latitude, longitude
Output: Tree Partitioning in different servers
(#Locations to start with - 500 $\mathcal{M}$)

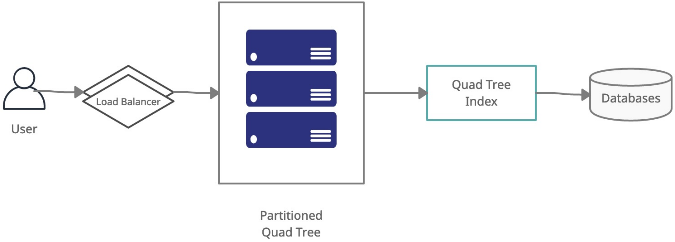
1. Divide the servers based on locations
2. Hash the locations and IDs and map them with different servers where they will be stored
3. Query all servers to return nearby places for a location
4. Return the values to the user

PopularPlacesFinder($Places$) ▷ Popular places nearby

Input: locationID, latitude, longitude
Output: Find most popular places within a given radius
(Lets assume we keep track of popularity of each place)

1. Store the popularity number in database as well as tree
2. Iterate through partitions and return top 50 places for each server
3. Return the places queried by the required server

Step 9. Service diagram



Step 10. Business model

- Advertisements
- Commissions from partnerships

References

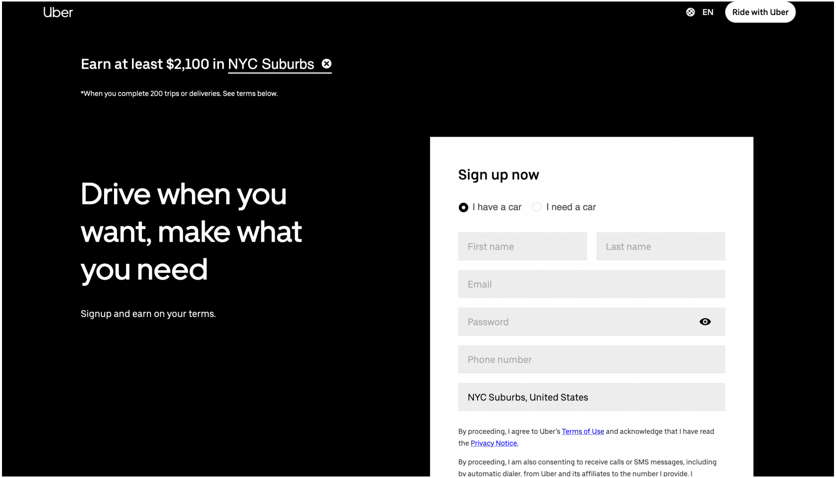
- Yelp/Nearby friends Design in Grokking the System Design Interview
- Yelp/Nearby by Astik Anand
- System Design Tutorial- Yelp

Uber HOME

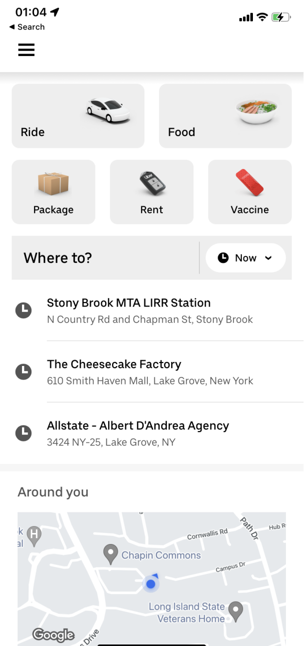
Step 1. Problem

- Design a simple Uber like service where drivers use their personal cars to drive customers around. Both customers and drivers communicate through their smartphones

Step 2. User Interface



Step 2. User Interface



Step 3. Requirements

Functional requirements

- Customers can request a ride
- Drivers can accept or reject the ride request
- Once accepted, share customer's and driver's location with each other until the trip is ended

Non-functional requirements

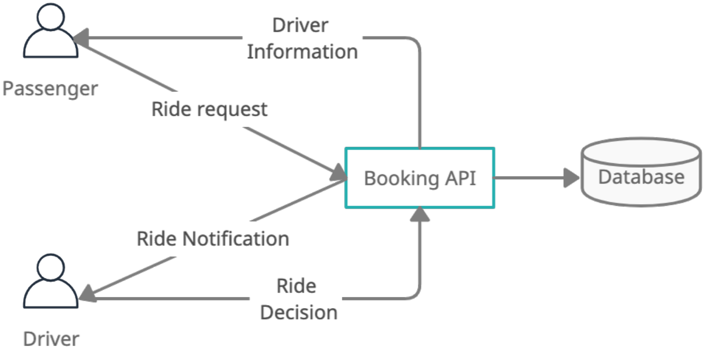
- Heavy search load
- Low latency
- Security

Step 4. Statistics

Feature	Assumptions
Drivers	1 $\mathcal{M}$, 500k/day
Passengers	300 $\mathcal{M}$, 1 $\mathcal{M}$/day
Rides	1 $\mathcal{M}$/day
Driver location update	every 3 secs

Parameter	Estimation(Assuming Quadtree)
Tree storage	1 $\mathcal{M} \times 8 \times 3$ bytes $=$ 24 MB
Driver hash table storage	1 $\mathcal{M} \times 35$ bytes $=$ 35 MB
Subscription storage	(500k $\times$ 3 bytes) $+$(500k $\times$ 5 subscribers $\times$ 8 bytes) $=$ 21 MB
Location upload (Driver side)	5 $\times$ 500k $=$ 2.5 $\mathcal{M}$
Location upload bandwidth (Driver side)	2.5 $\mathcal{M} \times$ (3+16) bytes $=$ 47.5 MBps

Step 5. Block diagram



Step 6. Database schema

Drivers	
Driver ID(PK)	int
Name	varchar(256)
Old position	decimal
Current position	decimal
Ratings	int

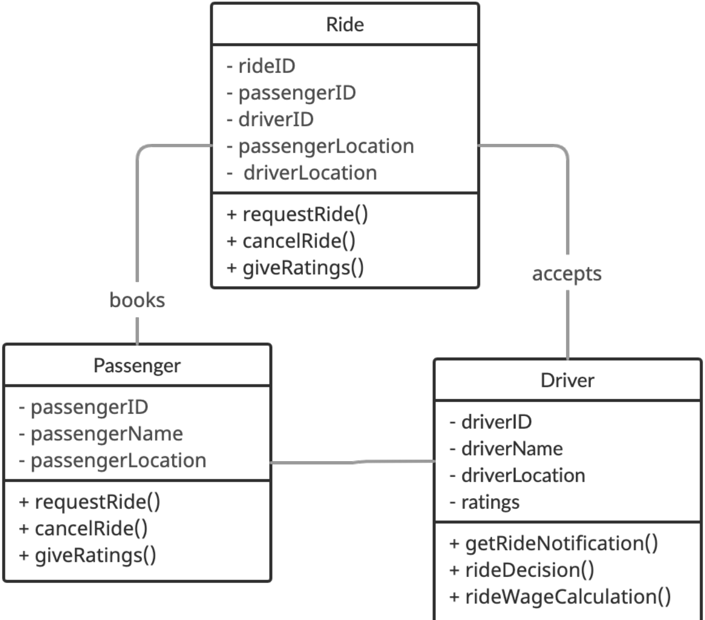
Passengers	
Passenger ID(PK)	int
Name	varchar(256)
Current position	int

Step 6. Database schema

- Metadata

Table	Storage estimation
Drivers	DriverID (3 bytes) + Name (256 bytes) + Old position (16 bytes) + Current position (16 bytes) + Ratings (3 bytes) = 294 bytes 1 $\mathcal{M}$ $\times$ 299 bytes = 0.3 GB
Passengers	PassengerID (3 bytes) + Name (256 bytes) + Current position (16 bytes) = 275 bytes 300 $\mathcal{M}$ $\times$ 275 bytes = 82.5 GB

Step 7. Class diagram



Step 8. Algorithm

$\textsc{PushModel}(Drivers)$ ▷ Broadcast driver location from hash tree

Input: DriverID, Old position, Current position, PassengerID
Output: Refresh quad tree server as the driver location is changed and broadcast the location to subscribers
(#Drivers - 1 $\mathcal{M}$, 500k active/day)

1. Query the server to find nearby drivers as the passenger opens the app
2. Subscribe the passenger for all updates from the list of nearby drivers
3. Maintain a list of interested passengers
4. Send notification to the passengers in list every time the hash tree is updated

Step 8. Algorithm

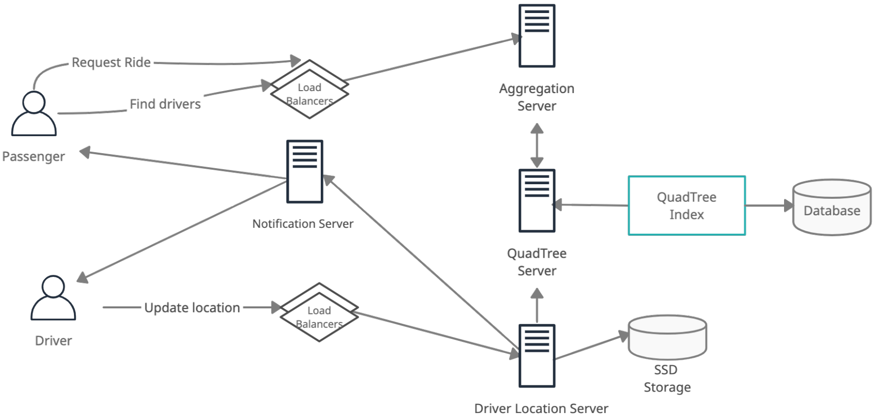
$\textsc{RequestRide}(Drivers, Passengers)$ ▷ Request a ride

Input: Driver ID, Old position, Current position, Passenger ID, Current position

Output: Return a driver for the requested ride

1. Aggregator server takes the request and asks quadtree servers to return a list of nearby drivers
2. Collects all results and sorts them by ratings
3. Send notification to top three drivers from the list
4. Send notification for driver who accepted the ride first and cancel for all others
5. If none of them accepts the ride, send a notification to next three drivers from the list

Step 9. Service diagram



Step 10. Business model

- Commission-based model

References

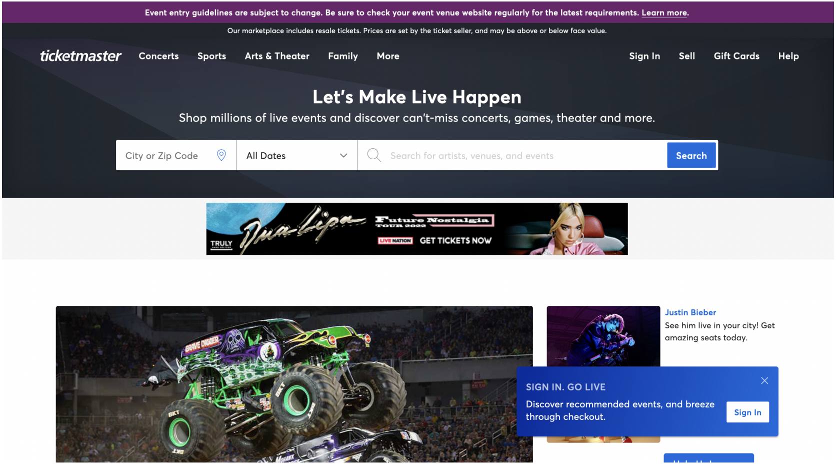
- Uber backend Design in Grokking the System Design Interview
- Uber System Design by Naren Gowda
- System Design of Uber App- GeeksForGeeks
- Uber System Design- codeKarle

Ticketmaster HOME

Step 1. Problem

- Design an online ticketing system that sells movie tickets like Ticketmaster or BookMyShow

Step 2. User Interface



Step 3. Requirements

Functional requirements

- Location-based queries on movies, shows and theatres
- Book tickets for a particular show

Non-functional requirements

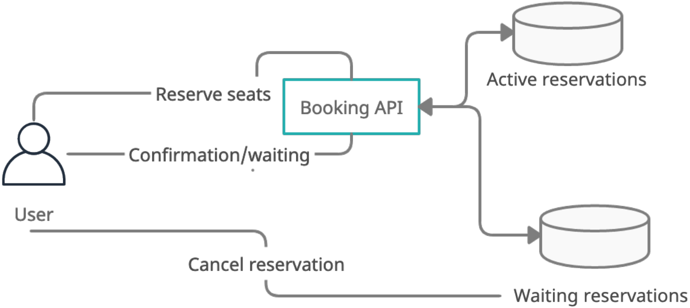
- High concurrency
- Low latency
- Security

Step 4. Statistics

Feature	Assumptions
Page views	3 $\mathcal{B}$/month
Tickets sold	10 $\mathcal{M}$/month
Cities	500
Theatres	10/city
Shows	2 shows per movie per theatre
Seats	2000
Seat booking	100 bytes (IDs, NumberOfSeats, ShowID, MovieID, etc)
Storage time	5 years

Parameter	Estimation
Storage	$500 \times 10 \times 2000 \times 2 \times (100+100)$ bytes $= 4$ GB/day
Monthly download volume	$3\ \mathcal{B} \times 100$ bytes $= 0.1$ TB
Download bandwidth	0.1 TB $\times$ 24 / 3600/ 30 secs $=$ 22.2 MBps

Step 5. Block diagram



Step 6. Database schema

User	
User ID(PK)	int
Name	varchar(256)
Password	varchar(20)
Email	varchar(256)
Phone	varchar(16)

Theatre	
Theatre ID(PK)	int
Total seats	int
City	varchar(90)
State	varchar(90)
Zip code	int

Booking	
Booking ID(PK)	int
Timestamp	datetime
No. of seats	int
User ID	int
Show ID	int
Status	int (enum)
Transaction ID	int

Show	
Show ID(PK)	int
Movie ID	int
Start time	datetime
End time	datetime
Theatre ID	int

Step 6. Database schema

- Metadata

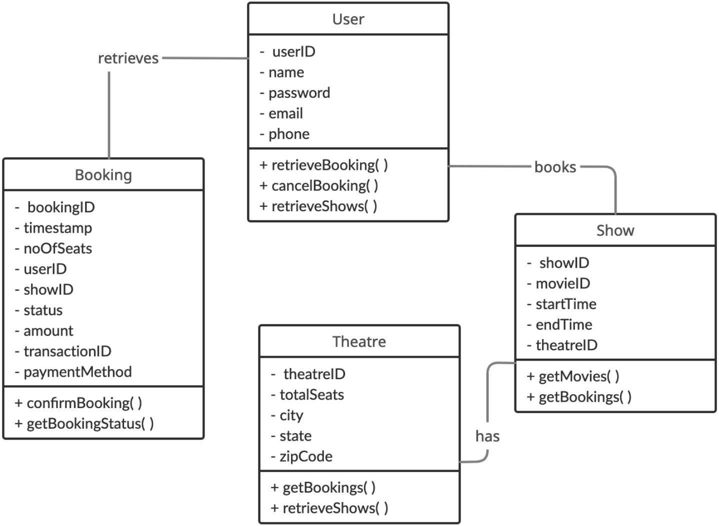
Table	Storage estimation
User	UserID (3 bytes) + Name (256 bytes) + Password (20 bytes) + Email (256 bytes) + Phone (16 bytes) = 551 bytes 1 $\mathcal{M}$ $\times$ 551 bytes = 0.5 GB
Theatre	TheatreID (3 bytes) + TotalSeats (3 bytes) + City (90 bytes) + State (90 bytes) + ZipCode (3 bytes) + TotalSeats (3 bytes) = 192 bytes 5000 $\times$ 192 bytes = 0.96 MB

Step 6. Database schema

- Metadata

Table	Storage estimation
Booking	BookingID (3 bytes) + Timestamp (8 bytes) + NoOfSeats (3 bytes) + UserID (3 bytes) + ShowID (3 bytes) + status (1 byte) + + TransactionID (3 bytes) = 28 500k × 28 bytes = 14 MB
Show	ShowID (3 bytes) + MovieID (3 bytes) + StartTime (8 bytes) + EndTime (8 bytes) = 22 bytes

Step 7. Class diagram



Step 8. Algorithm

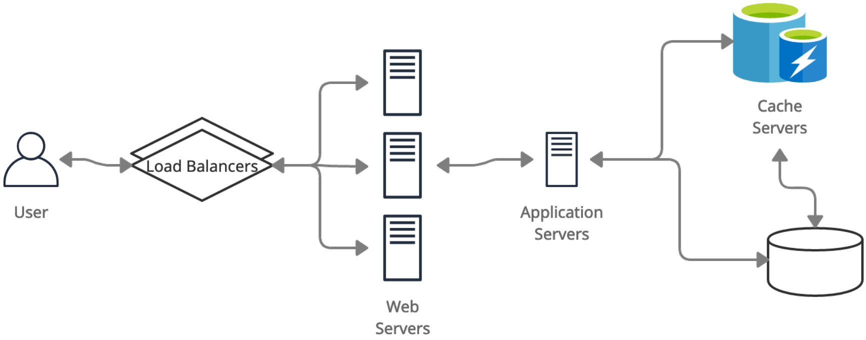
ACTIVERESERVATIONSERVICE($Show, Booking$) ▷ Active reservations

Input: UserID, BookingID, ShowID, Status

Output: Notify WaitingUsersService in case the booking is completed or reservation expires

1. Store all reservations of a show in linked hashmap
2. Head of hashmap points to the oldest reservation due to expiry time
3. Status field will have default value 1(Reserved), which gets updated to 2(Booked) as the booking is complete
4. When time is expired, it gets marked as 3(Expired) and the WaitingUsersService will get notified to serve waiting users

Step 9. Service diagram



Step 10. Business model

- Commission-based model

References

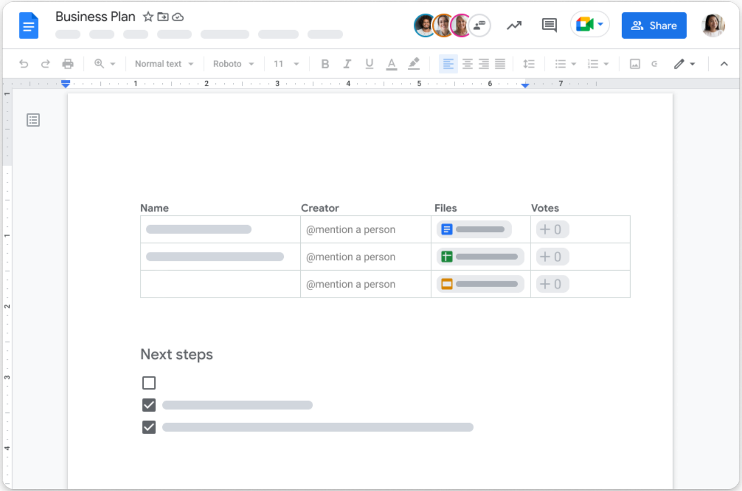
- Ticketmaster Design in Grokking the System Design Interview
- Systems Design: Ticketmaster

Google Docs HOME

Step 1. Problem

- Design a simple document collaboration tool like Google Docs, where users who have access to a document can create, edit, share documents online. Several users can edit the same document simultaneously.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Change a document at the same time without any conflict
- Give appropriate permission(view, editor) for a document
- Import/export/comment/annotate document

Non-functional requirements

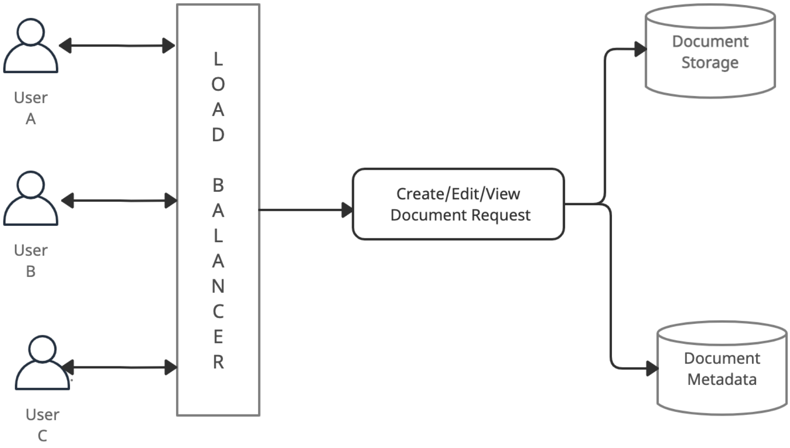
- High concurrency
- High consistency
- Low latency
- High availability

Step 4. Statistics

Feature	Assumptions
Users	$200\mathcal{M}$ total, $20\mathcal{M}$ daily active
#Documents/user	30
Average document size	200 KB
User activity	2 documents/day
Storage	5 years

Parameter	Estimation
Document space	$200\mathcal{M} \times 30 \times 200$ KB $=$ 1200 TB
Daily upload volume	$20\mathcal{M} \times 100$ KB $=$ 2 TB
Upload bandwidth	2 TB / (24×3600 sec) $=$ 23.15 MB/sec
Daily download volume	$20\mathcal{M} \times 2 \times 200$ KB $=$ 8000 GB
Download bandwidth	8000 GB / (3600×24 sec) $=$ 92.5 MB/sec

Step 5. Block diagram



Step 6. Database schema

User	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
LastActive	datetime

Document	
Document ID(PK)	int
Author ID	int
Creation time	datetime
Updation time	datetime
Document URL	varchar(256)
Title	varchar(256)

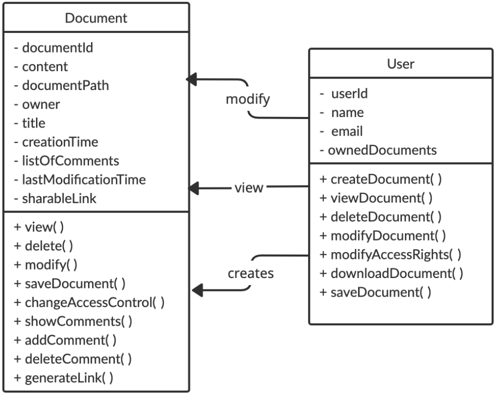
AccessControl	
Document ID	int
User ID	int
Access/Control type	varchar(20)

Step 6. Database schema

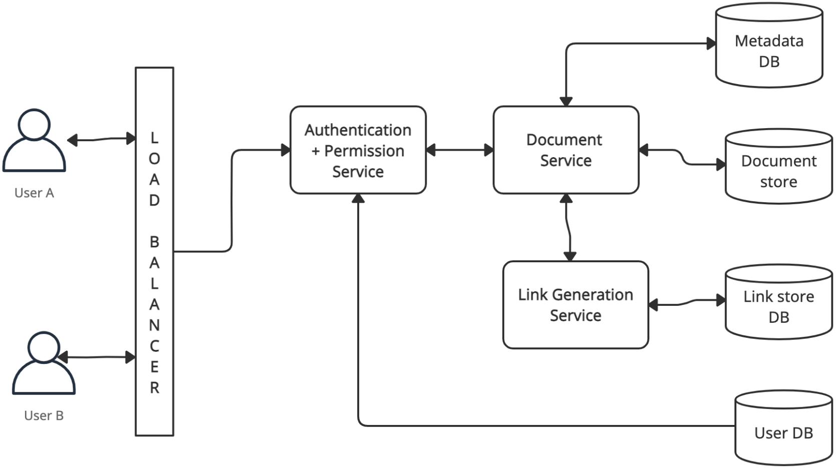
- Metadata

Table	Storage estimation
User	UserID (4 bytes) + Name (20 bytes) + Email (32 bytes) + LastActive (4 bytes) = 60 bytes 200M (users) × 60 bytes = 12 GB
Document	Document ID (4 bytes) + Author ID (4 bytes) + Creation timestamp (4 bytes) + Last modification (4 bytes) + Document URL (256 bytes) + Title (256 bytes) = 528 bytes $200\mathcal{M} \times 30 \times 528$ bytes = 3168 GB
AccessControl	Document ID (4 bytes) + User ID (4 bytes) + Access type (1 byte) = 9 bytes

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

EDITLOCALLY($change, position$) ▷ apply local changes to the document at that character position

Input: change(insert/delete/modify)
1. Iterate through the list of local changes
2.   Apply each change to the document at given position
3. Save new version at client side
4. Send changes with new version number to the server to be synced by other clients

SYNCCHANGES() ▷ Sync changes from all other clients through the server

1. Fetch the new changes from the server
2. Iterate through this list of changes
3.   Apply Operational Transformation on each change and apply the change
4. Save the new version locally

Step 10. Business Model

- Subscription fee

References

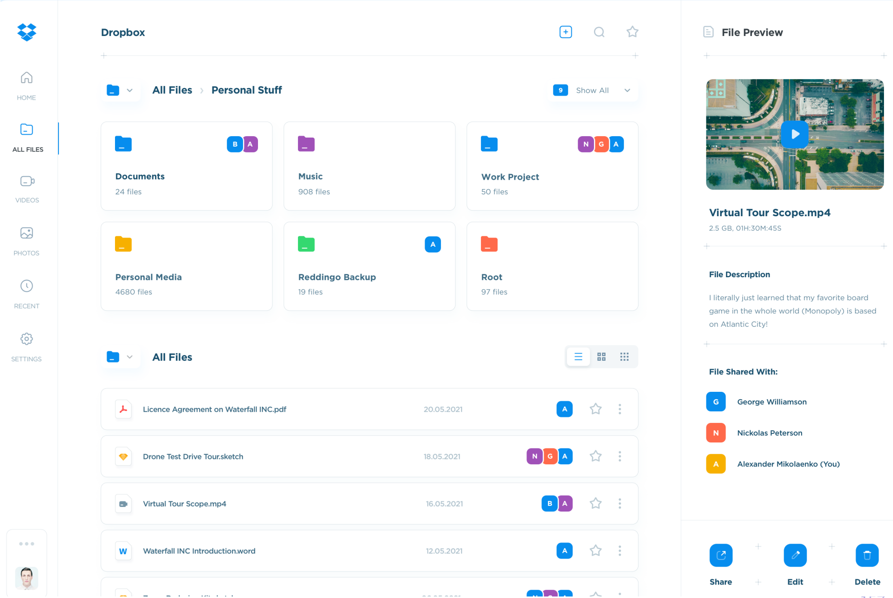
- Google Docs High-Level System design on LinkedIn
- Operational Transformation Youtube video by Tech Dummies
- Design Google Docs blog
- How Does Google Sheets work? Medium article
- Designing Google Docs by gainlo
- System design Google Docs on AlgoDaily

Dropbox HOME

Step 1. Problem

- Design a cloud file storage service like Dropbox or Google Drive which enable users to store their data on remote servers.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Upload and download their files/photos from any device
- Share files and folders with others
- Support automatic synchronization between devices
- Add,delete and modify files while offline

Non-functional requirements

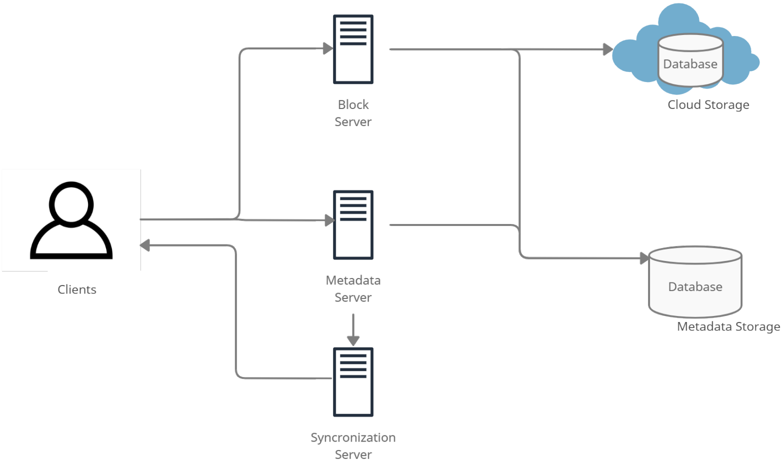
- High scalability
- High availability
- ACID-ity is required. Atomicity, Consistency, Isolation and Durability of all file operations should be guaranteed.

Step 4. Statistics

Feature	Assumptions
Users	$200\mathcal{M}$ total, $20\mathcal{M}$ daily active
#Documents/user	30
Average document size	200 KB
User activity	2 documents/day
Storage	5 years

Parameter	Estimation
Document space	$200\mathcal{M} \times 30 \times 200$ KB $=$ 1200 TB
Daily upload volume	$20\mathcal{M} \times 100$ KB $=$ 2 TB
Upload bandwidth	2 TB / (24×3600 sec) $=$ 23.15 MB/sec
Daily download volume	$20\mathcal{M} \times 2 \times 200$ KB $=$ 8000 GB
Download bandwidth	8000 GB / (3600×24 sec) $=$ 92.5 MB/sec

Step 5. Block diagram



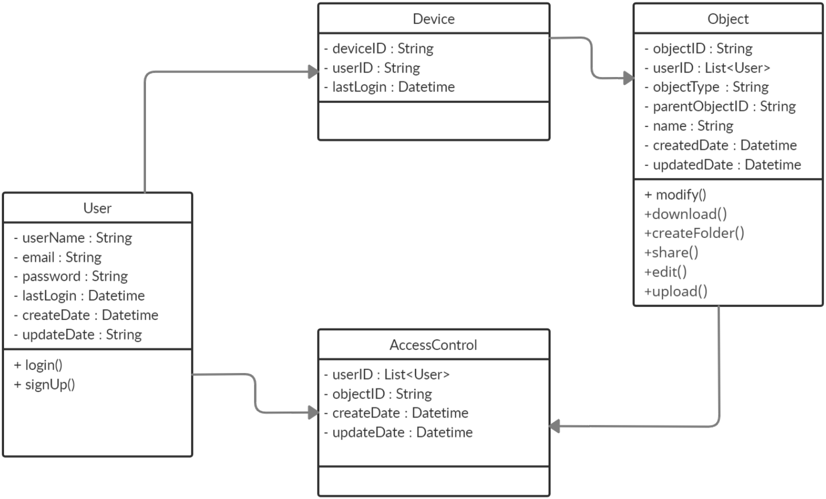
Step 6. Database schema

User	
User ID (PK)	int
Name	varchar(20)
Email	varchar(32)
Password	varchar(32)
Creation time	datetime
Update time	datetime
Last login	datetime

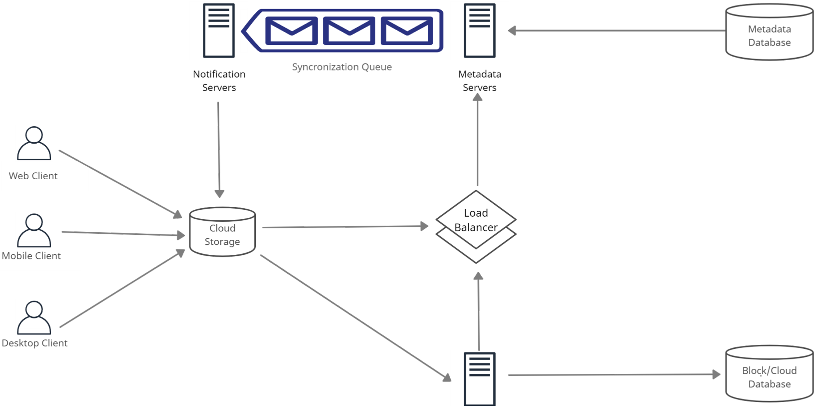
Devices	
Device ID (PK)	int
User ID	int
Creation time	datetime
Update time	datetime

Objects	
Object ID (PK)	int
User ID (PK,FK)	int
Object type	varchar(32)
Parent Object ID	int
Name	varchar(20)
Creation time	datetime
Update time	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

EDITLOCALLY($change, position$) ▷ apply local changes to the document at that character position

Input: change(insert/delete/modify)

1. Iterate through the list of local changes
2. Apply each change to the document at given position
3. Save new version at client side
4. Send changes with new version number to the server to be synced by other clients

SYNCCHANGES() ▷ Sync changes from all other clients through the server

1. Fetch the new changes from the server
2. Iterate through this list of changes
3. Apply Operational Transformation on each change and apply the change
4. Save the new version locally

Step 10. Business model

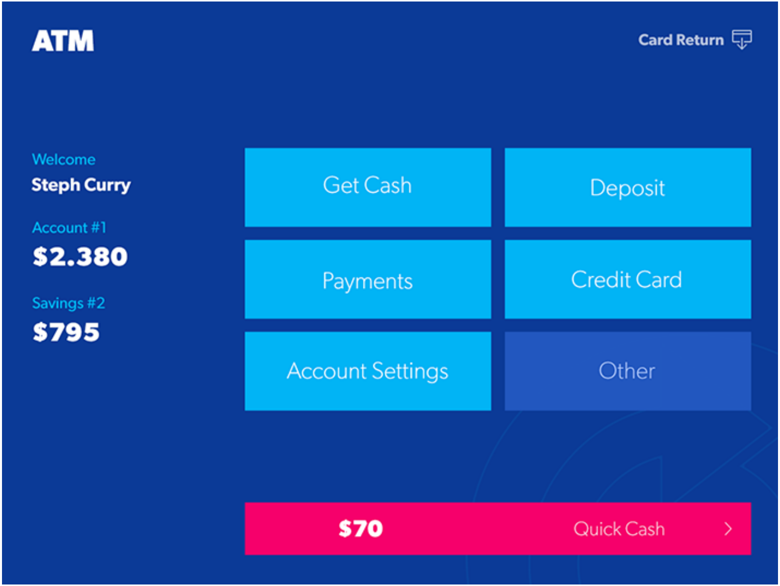
- Freemium

ATM HOME

Step 1. Problem

- Design an automated teller machine (ATM) that allows clients to access to financial transactions in a public space without the need for a cashier.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Dispense/Deposit cash
- Display current balance

Non-functional requirements

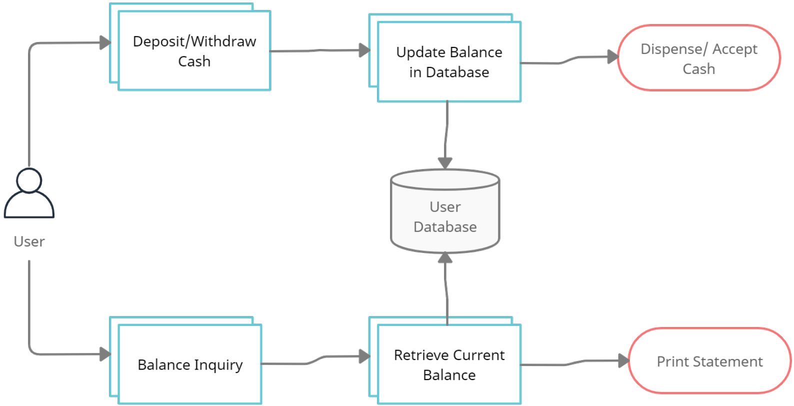
- High reliability
- ACID-ity is required. Atomicity, Consistency, Isolation and Durability of all file operations should be guaranteed.

Step 4. Statistics

Feature	Assumptions
Daily average users	1000
Daily transactions	2000
Max cash capacity	2000000
Max withdrawal amount per user	1000

Parameter	Estimation
Daily average withdrawal amount	1000000
Size of one transaction	20 KB
Average network bandwidth	10 MB/sec
Latency	5 sec

Step 5. Block diagram

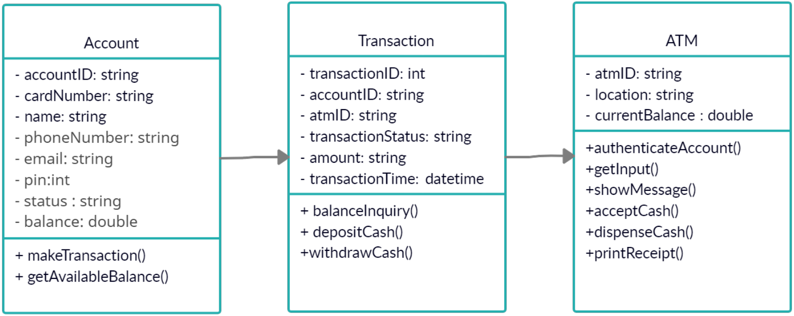


Step 6. Database schema

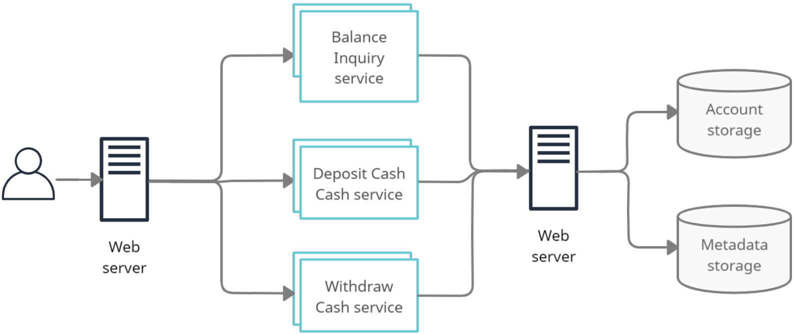
Account	
Account ID (PK)	int
Card Number (PK)	int
Name	varchar(20)
Email	varchar(32)
PIN	int
Status	varchar(20)
Balance	double

Transaction	
Transaction ID (PK)	varchar(32)
Account ID (FK)	int
ATM ID (PK)	int
Transaction Status	varchar(32)
Amount	double
Transaction time	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

AuthenticateUser(card,PIN)

Input: card and PIN
Output: Check if the user is valid or not
1. Use card reader service to get account information of the user
2. Provide transaction options
3. Eject card

WithdrawCash(account,cash)

Input: account and cash
Output: Cash
1. If entered amount is less than balance
2. dispense cash and update balance
3. Display message if transaction was successful or not

DepositCash(account,cash)

Input: account and cash
Output: Update message
1. Calculate cash deposited
2. Update account balance and ATM balance
3. Display message if transaction was successful or not

Step 10. Business model

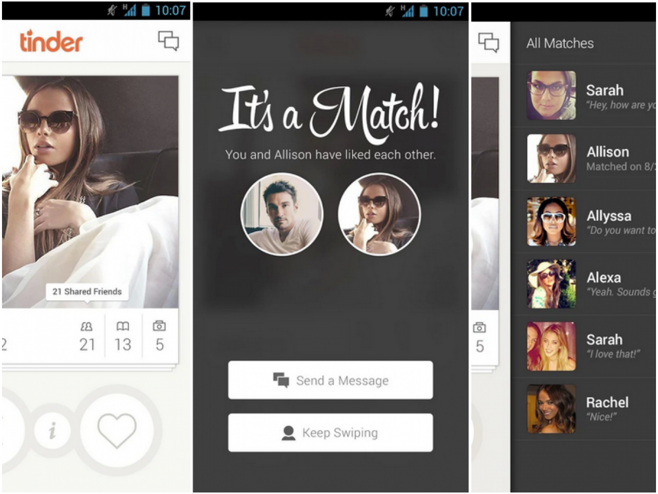
- Transaction charges
- Reduction in interest rates

Tinder HOME

Step 1. Problem

- Design an online dating application which allow users to use a swiping motion to like (swipe right) or dislike (swipe left) and match and chat with other users .

Step 2. User interface



Step 3. Requirements

Functional requirements

- Swipe left (dislike) and right (like)
- If both account like each other, match profiles
- Add user profile
- Chat option
- Super likes

Non-functional requirements

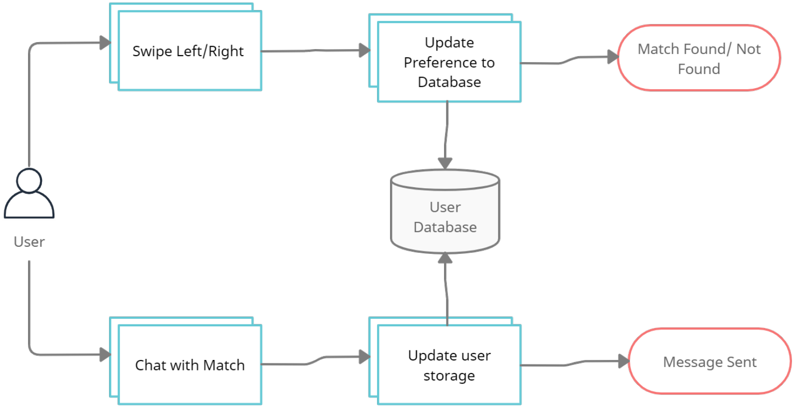
- Minimum latency
- High availability
- High scalability
- High consistency
- High reliability

Step 4. Statistics

Feature	Assumptions
Users	50 $\mathcal{M}$
Daily matches	1 $\mathcal{M}$
Daily users	10 $\mathcal{M}$
Daily swipes	50 $\mathcal{M}$
Number of images per user	5

Parameter	Estimation
Image space	200 KB
Description space	140 KB
Total storage	50 $\mathcal{M} \times (5 \times 200\text{KB} + 150\text{KB}) = 53$ PB

Step 5. Block diagram

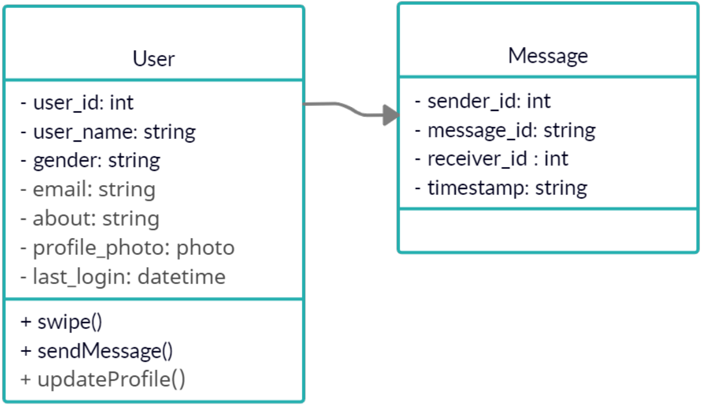


Step 6. Database schema

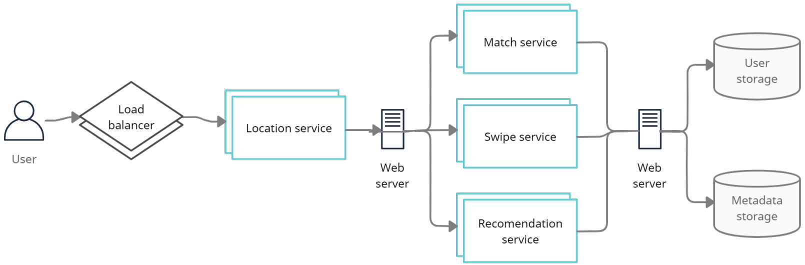
User	
User ID (PK)	int
User name	varchar(32)
Gender	varchar(32)
Email	varchar(32)
About	varchar(140)
Profile photo	blob
Match radius	int
Last login	datetime

Message	
Message ID (PK)	int
Sender ID	int
Receiver ID	int
Timestamp	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

RECOMMEND POTENTIAL MATCHES(user)

Input: User preferences like location, gender, age and match radius
Output: Recommend potential matches to swipe based on user's preference

1. Divide the world map into boxes using geosharding
2. Size of each box is dependent on user count and active user count
3. Based on location preference, assign a box to user
4. Based on match radius preference determine potential matches from user and neighbour boxes
5. **for** each neighbor in neighbor boxes **do**
6. if user is within match radius
7. add neighbor to user recommendation
8. Sort recommendation based on user preferences
9. User receives top recommended people from neighborhood as a result

Step 10. Business model

- Premium Subscriptions

Video Streaming

- Video quality and resolution directly affect the bitrate and data used in streaming.
- Video quality is defined by the resolution (no. of pixels). Eg. 144p video has 256 $\times$ 144 pixels.
- Each pixel has a size of 3 bits. Therefore size of 1 frame s is given by :

$$s = 3 \times h \times w \ bits$$

- Given video quality, i.e, the resolution in height h and width w the rate to transfer 30 frames in 1 second (in Kbps) is :

$$bitrate = (3 \times h \times w \times 30)/1024 \ Kbps$$

Video Streaming (Contd.)

- Given actual bitrate, we can calculate frames transimted per second (f):

$$f = \frac{actual\ bitrate}{required\ bitrate} \times 30$$

- Time to transmit 30 frames (t) is given by

$$t = \frac{30}{f}\ seconds$$

- Delay is given by

$$delay = 1 - t$$

Video Streaming (Contd.)

- Eg for 144p resolution video height h = 144 and width w = 256. Required bitrate is

$$bitrate = (3 \times h \times w \times 30)/1024 = 3240\ Kbps$$

- If actual bitrate is 3000 Kbps, then frames transmitted per second (f) is given by

$$f = \frac{3000}{3240} \times 30 = 27.77\ fps$$

- Time to transmit 30 frames (t) is given by

$$t = \frac{30}{f} = 1.08\ seconds$$

- Delay is given by

$$delay = 1 - t = 0.08\ seconds$$

Netflix HOME

Step 1. Problem

- Netflix a video streaming service over the internet that allows users to stream and watch videos which are available on its platform.

Step 2. User interface

Step 3. Requirements

Functional requirements

- Create account and subscribe for a plan
- Search/play video
- Recommend videos

Non-functional requirements

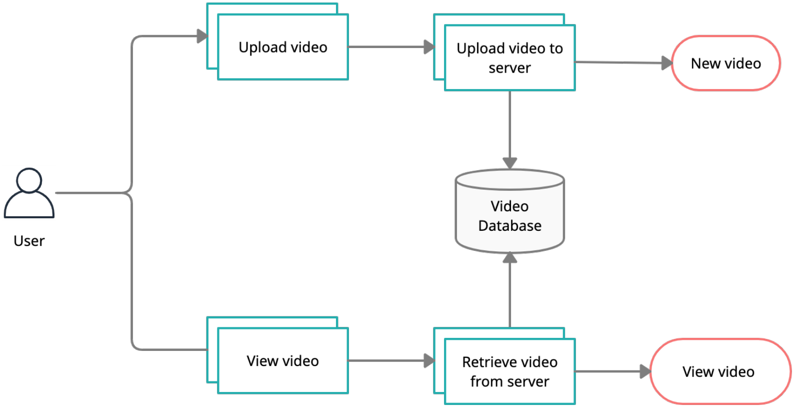
- High reliability
- High availability
- Minimal latency

Step 4. Statistics

Feature	Assumptions
Daily users	100 $\mathcal{M}$ users
Daily views	5 videos/user = 500 $\mathcal{M}$ views/day
Average video size	500 MB
Average number of uploads size	1000/day

Parameter	Estimation
Outgoing bandwidth	500 $\mathcal{M}$ $\times$ 500 MB = 250 PB/day
Incoming bandwidth	1000 $\times$ 500 MB = 500 GB/day
Storage	500 GB $\times$ 5 years $\times$ 365 days = 913 TB

Step 5. Block diagram



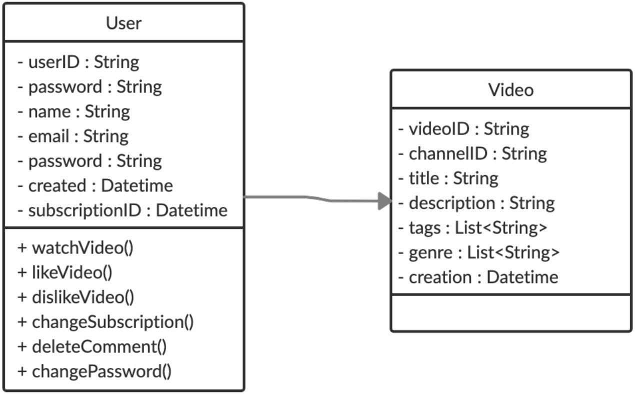
Step 6. Database schema

User	
User ID (PK)	int
Name	varchar(140)
Email	varchar(140)
Password	varchar(30)
Created	datetime
Last login	datetime
Subscription ID	int

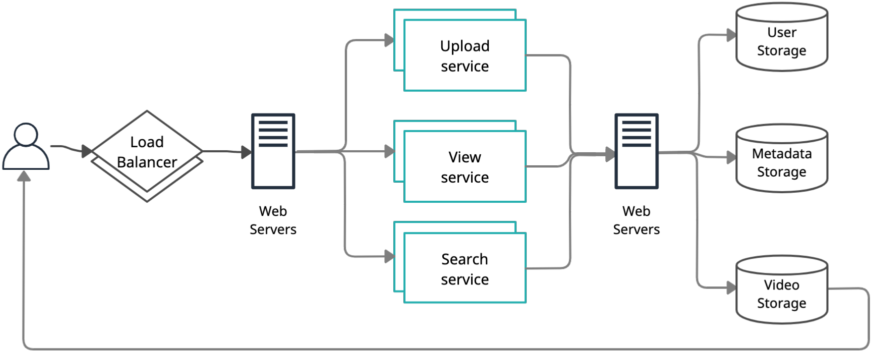
Video	
Video ID (PK)	int
Title	varchar(140)
Summary	varchar(140)
URL	varchar(140)
Length	int
Censor rating	datetime
Created	datetime

Subscription	
Subscription ID (PK)	int
User ID (FK)	int
Plan ID (FK)	int
Email	varchar(140)
Valid till	datetime
Created	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

RECOMMEND(user)

Input: User's watch history, subscriptions, likes and dislikes
Output: Recommend videos personalised for the user

1. Rank videos based on the satisfaction metrics
2. View history data (change over time)
3. Popularity, likes, dislikes
4. Trending shows, seasonality and competition
5. User receives top ranked videos as recommendation

Step 10. Business model

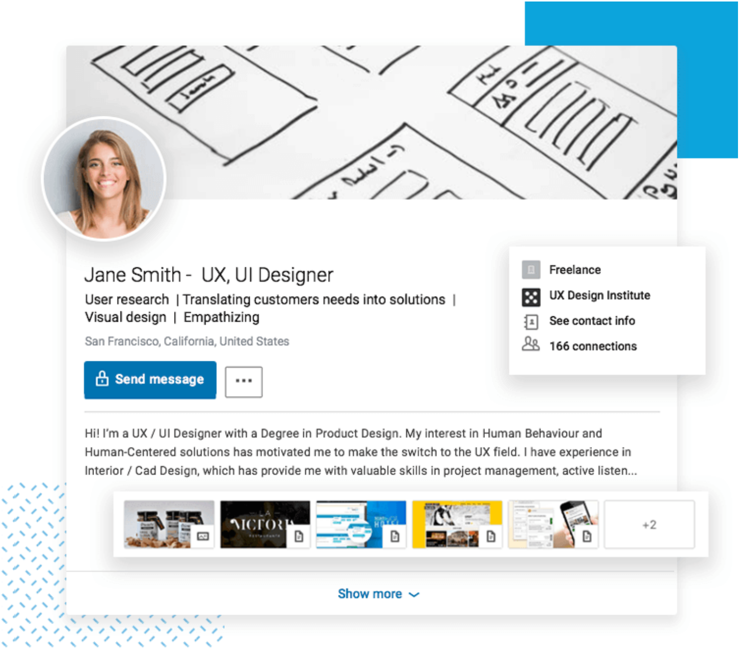
- Paid Subscriptions

LinkedIn HOME

Step 1. Problem

- LinkedIn is a professional social network that allows users to connect and explore new opportunities for their careers.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Create profile
- Connect with other users
- Create/Apply to job postings

Non-functional requirements

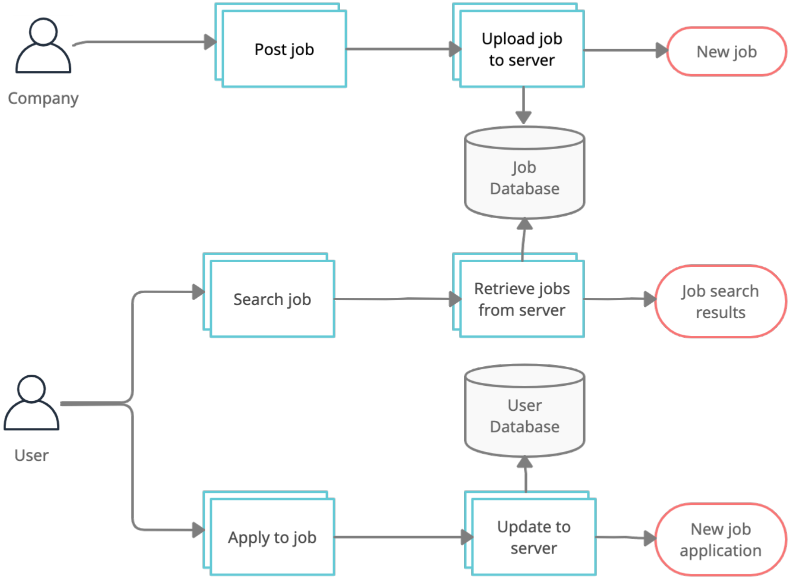
- High availability
- Partition tolerant
- Eventual consistency

Step 4. Statistics

Feature	Assumptions
Daily users	100 $\mathcal{M}$ users
Daily job postings	5 $\mathcal{M}$
Daily job applications	2 $\mathcal{M}$
Average job posting size	2 KB
Total companies	57 $\mathcal{M}$

Parameter	Estimation
Job posting space	5 $\mathcal{M}$ $\times$ 2 KB $=$ 10 GB/day
Total bandwidth	10 GB/day
Storage	10 GB $\times$ 5 years $\times$ 365 days $=$ 18 TB

Step 5. Block diagram



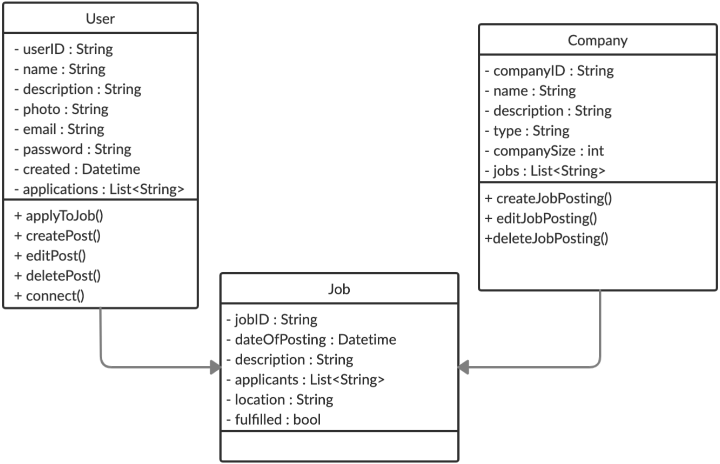
Step 6. Database schema

User	
User ID (PK)	int
Name	varchar(140)
Description	varchar(140)
Photo	blob
Email	varchar(140)
Password	varchar(30)
Created	datetime
Last login	datetime
Applications	varchar(1000)

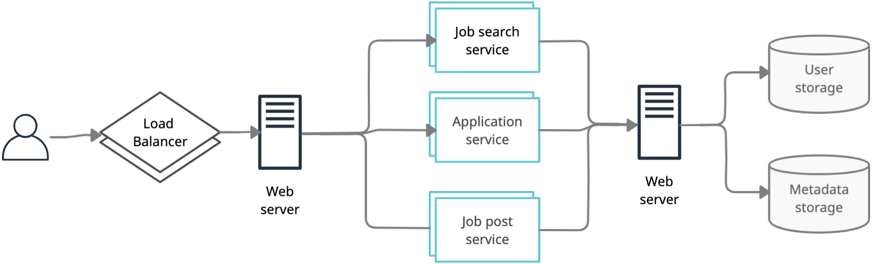
Company	
Company ID (PK)	int
Name	varchar(140)
Description	varchar(450)
Type	varchar(140)
Size	int
Jobs	varchar(1000)

Job	
Job ID (PK)	int
Date of posting	datetine
Description	varchar(140)
Location	varchar(140)
Applicants	vacrchar(1000)
Fulfilled	bool

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

JobRecommendation(user)

Input: Candidate's profile

Output: Recommend job postings personalised for the candidate

1. Use decision trees and deep learning to recommend jobs based on :
2. Work similarity
3. Experience/Skills
4. Location
5. Likelihood of response
6. User receives top ranked job postings as recommendation

Step 10. Business model

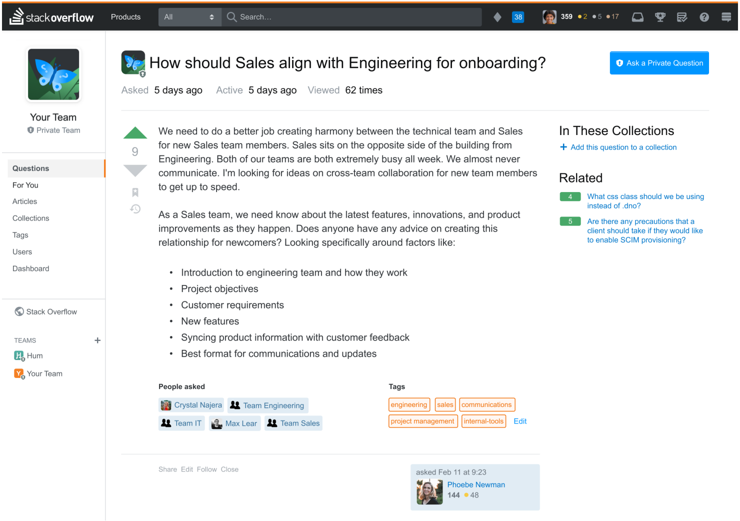
- Premium Subscriptions

Stack Overflow HOME

Step 1. Problem

- Stack Overflow is one of the largest online communities for developers to learn and share their knowledge. The website provides a platform for its users to ask and answer questions.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Search/Post/Answer questions
- Add comments to questions or answers
- Upvote/Downvote anwsers

Non-functional requirements

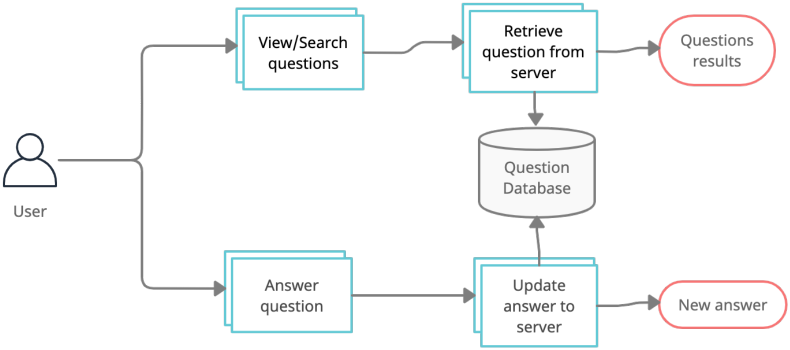
- Minimum latency
- High availability

Step 4. Statistics

Feature	Assumptions
Daily users	10 $\mathcal{M}$ users
Daily questions	7600
Daily questions answered	5320
Average number of answers for each question	5
Average size of question/answer	30 KB

Parameter	Estimation
Question space	$(1 + 5) \times 7600 \times 30$ KB $= 1.3$ GB/day
Total bandwidth	1.3 GB/day

Step 5. Block diagram



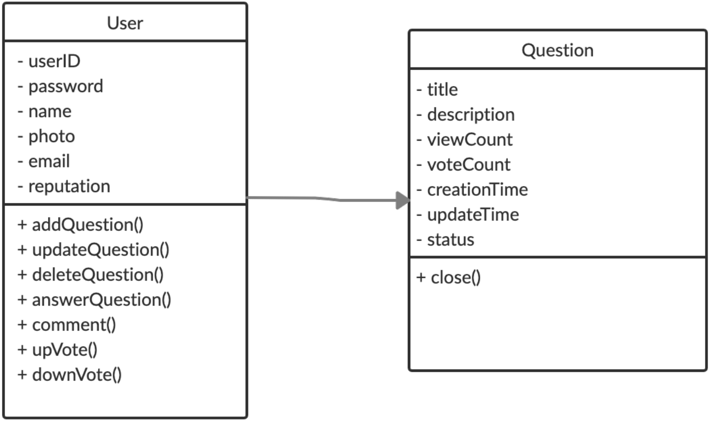
Step 6. Database schema

User	
User ID (PK)	int
Password	varchar(140)
Name	varchar(140)
Email	blob
Reputation	varchar(140)
Created	datetime
Last login	datetime

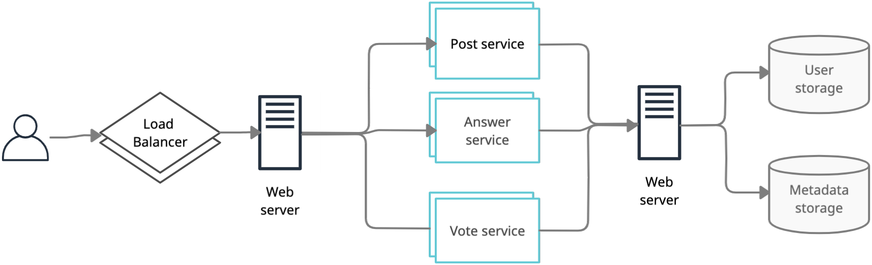
Question	
Question ID (PK)	int
Title	varchar(140)
Description	varchar(450)
View count	int
Vote count	int
Creation time	datetime
Update time	datetime
Status	varchar(140)

Answer	
Answer ID (PK)	int
Answer text	varchar(500)
Accepted	bool
Vote count	int
Create time	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

SEARCH(query)

Input: User query intended search similar questions
Output: Retrieve existing questions that closely resembles user's query

1. Tag prediction - Predict which tag the query best belongs to
2. Tag classifier is trained using an LSTM model to get word embeddings
3. Information Retrieval - Use word embedding and cosine distance to retrieve similar existing questions
4. User receives top ranked questions as a search result

Step 10. Business model

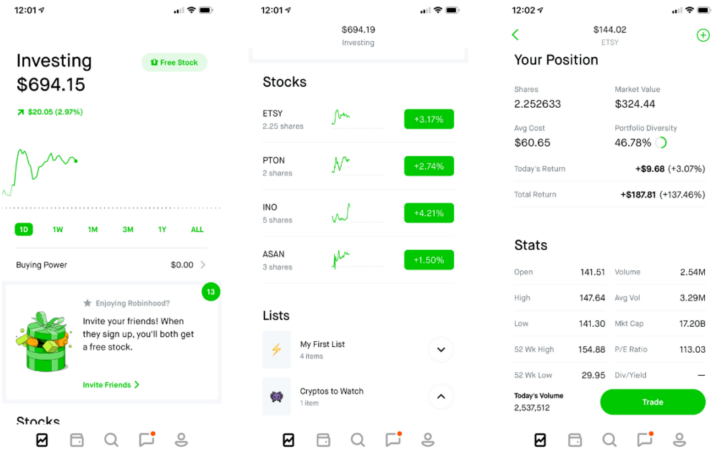
- Advertisements
- Job listings
- CV search

Robinhood HOME

Step 1. Problem

- An Online Stock Brokerage System like Robinhood facilitates its users the trade (i.e. buying and selling) of stocks online. It allows clients to keep track of and execute their transactions, and shows performance charts of the different stocks in their portfolios.

Step 2. User interface



Step 3. Requirements

Functional requirements

- Buy/sell stocks
- Add stocks to watchlist
- View real time and historic view on stock price reports

Non-functional requirements

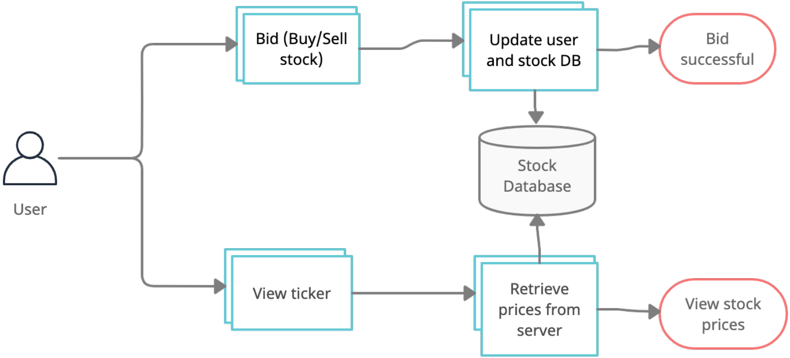
- Minimum latency
- High availability
- High reliability

Step 4. Statistics

Feature	Assumptions
Daily bids	100 $\mathcal{M}$ bids
Daily clients	10 $\mathcal{M}$
Number of tickers shown per client	10
Number of clients at a time	100k

Parameter	Estimation
Number of bids per second	100 $\mathcal{M}$ / $(8 \times 60 \times 60) =$ 3240 queries/sec
Bidding bandwidth	3420×200 bytes $= 0.6$
Ticker bandwidth	100k $\times$ 10 $\times$ 200 bytes $=$ 190 MB/sec
Total bandwidth	191 MB/sec

Step 5. Block diagram



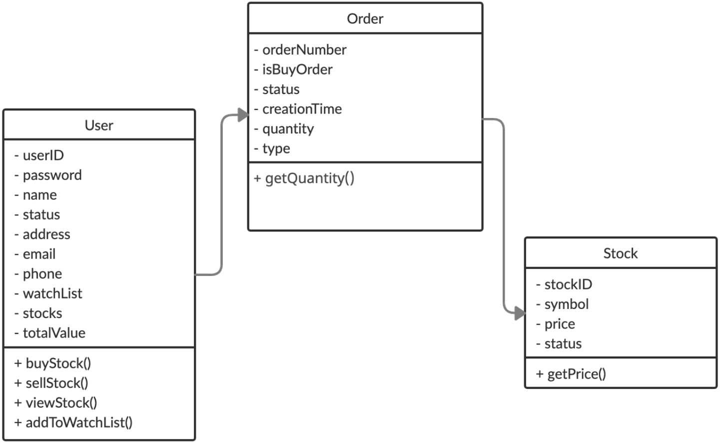
Step 6. Database schema

User	
User ID (PK)	int
Password	varchar(140)
Name	varchar(140)
Status	varchar(140)
Email	varchar(140)
Watchlist	varchar(1000)
Stocks	varchar(1000)
Total value	int

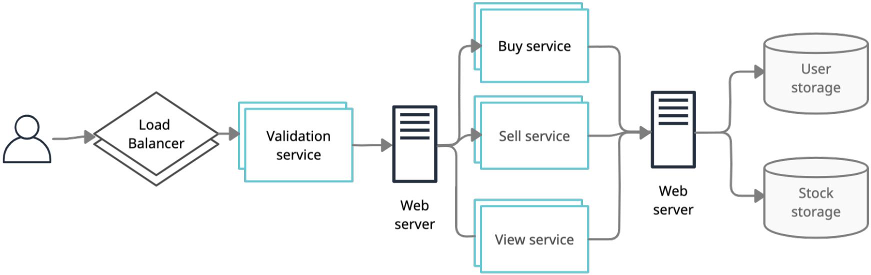
Question	
Question ID (PK)	int
Title	varchar(140)
Description	varchar(450)
View count	int
Vote count	int
Creation time	datetime
Update time	datetime
Status	varchar(140)

Answer	
Answer ID (PK)	int
Answer text	varchar(500)
Accepted	bool
Vote count	int
Create time	datetime

Step 7. Class diagram



Step 8. Service diagram



Step 9. Algorithm

SEARCH(query)

Input: User query intended search similar questions
Output: Retrieve existing questions that closely resembles user's query

1. Tag prediction - Predict which tag the query best belongs to
2. Tag classifier is trained using an LSTM model to get word embeddings
3. Information Retrieval - Use word embedding and cosine distance to retrieve similar existing questions
4. User receives top ranked questions as a search result

Step 10. Business model

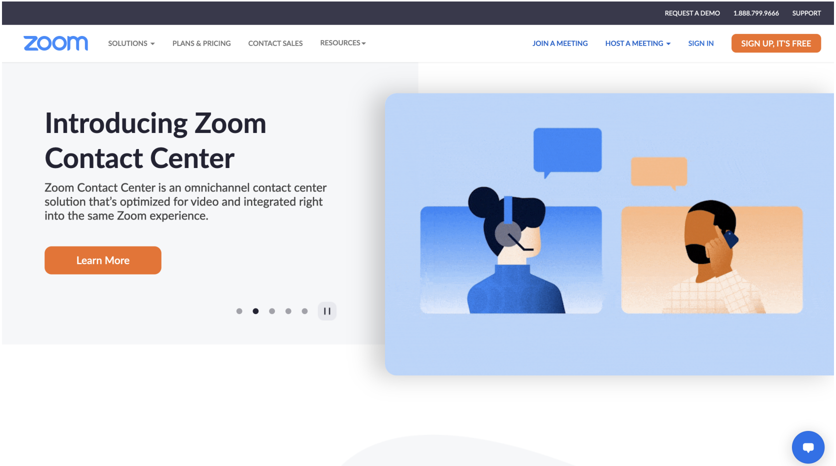
- Transaction-based revenues

Zoom HOME

Step 1. Problem

- Design an online video conferencing platform like Zoom

Step 2. User Interface



Step 3. Requirements

Functional requirements

- Supports 1-on-1 calls and group calls
- Calls can be audio, video or screen sharing and can be recorded

Non-functional requirements

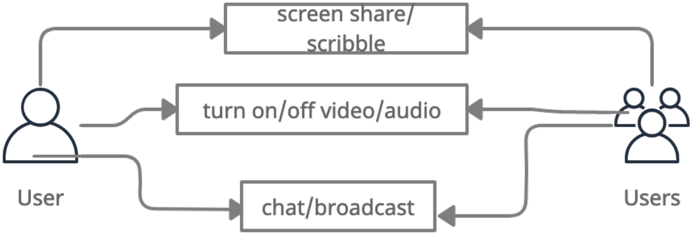
- Super fast
- Highly available
- Data loss is OK

Step 4. Statistics

Feature	Assumptions
Meetings	10 $\mathcal{M}$/day
Participants	300 $\mathcal{M}$/day
Participants per meeting	50-100
No. of Data centers	20

Parameter	Estimation
High quality download and upload bandwidth	100+100= 200 MBph

Step 5. Block diagram

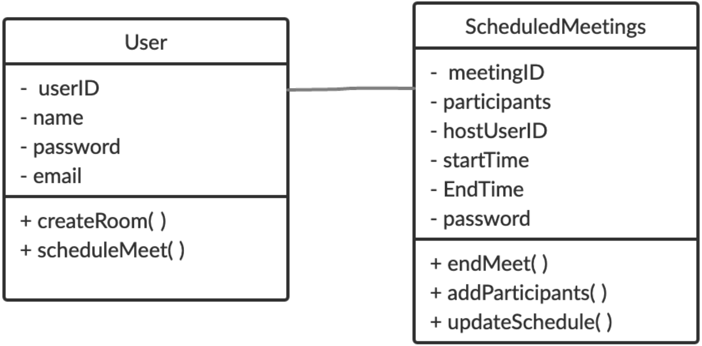


Step 6. Database schema

User	
User ID(PK)	int
Name	varchar(256)
Password	varchar(20)
Email	varchar(256)
IsMuted	bool
VideoStatus	bool
IsPresenting	bool

Scheduled Meetings	
Meeting ID(PK)	int
Participants	varchar(1024)
HostUserID	int
StartTime	datetime
EndTime	datetime
Password	varchar(20)
IsScreenshareEnabled	bool
BreakoutRooms	int
ChatMessage	varchar(1024)
IsScreenRecording	bool

Step 7. Class diagram



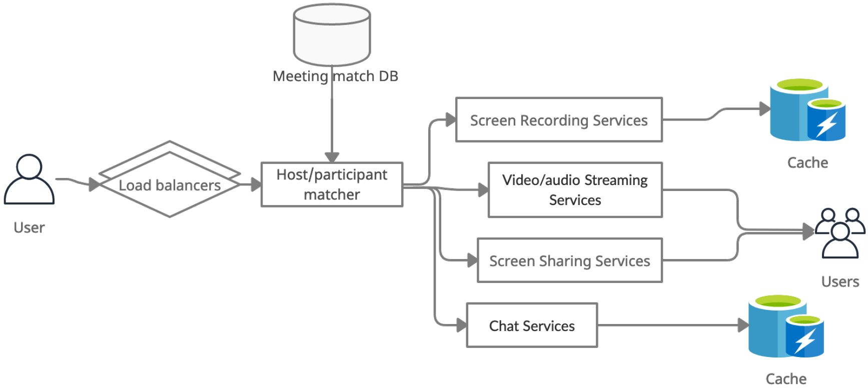
Step 8. Algorithm

STREAMING PACKETS($User$) ▷ Stream audio and video

Input: User ID, packets, Meeting ID, Participants, IPs
Output: Encoding audio and video stream and sending to the endpoint

1. If it is a one-on-one call, use peer to peer
2. Encode single stream with multiple layers
3. Perform video processing at client side and use UDP, TCP, TLS and HTTP for network layer
4. Divide participants based on their IP and the data center they fall in
5. At server side, determine optimal path to connect to participants with multimedia router
6. Perform video and audio processing at endpoint to enhance quality

Step 9. Service diagram



Step 10. Business model

- Freemium model
- Advertising

References

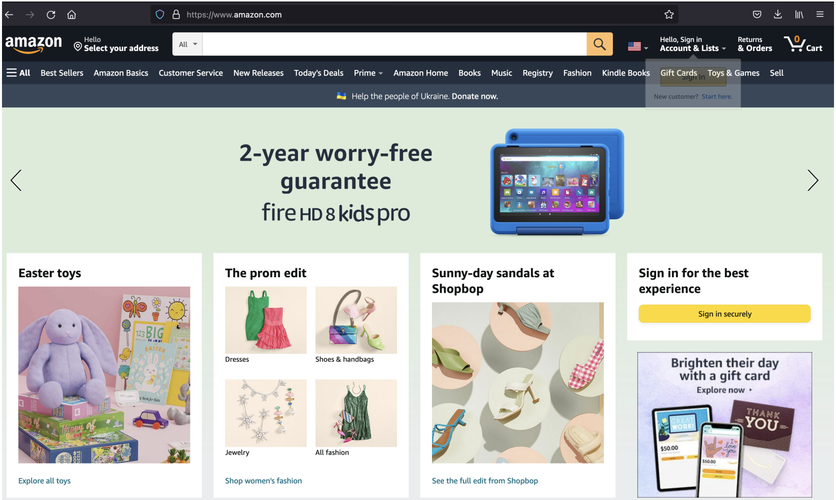
- How Zoom works
- Zoom System Design by codeKarle
- A Study of Zoom's Video Conferencing Architecture & System Design

Amazon HOME

Step 1. Problem

- Design an e-commerce website like Ebay/Amazon

Step 2. User Interface



Step 3. Requirements

Functional requirements

- Buyers can search products by name, keyword or category
- Buyers can add, update or delete products from their cart
- Sellers can add, modify or delete the products they want to sell
- Buyers can review and rate purchased products

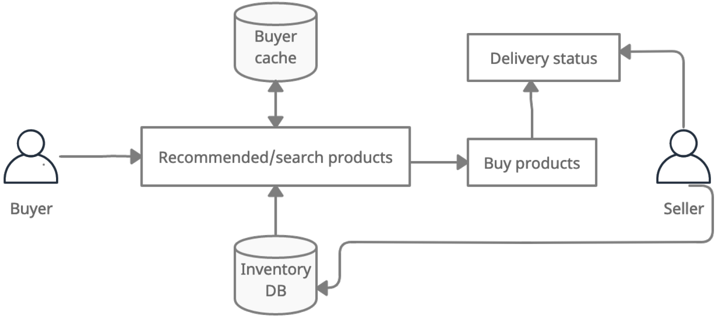
Non-functional requirements

- Highly available
- Low latency
- Highly consistent

Step 4. Statistics

Feature	Assumptions
Visits	2.3 $\mathcal{B}$/month
Sellers	9 $\mathcal{M}$
Items sold	1.6 $\mathcal{M}$/day

Step 5. Block diagram



Step 6. Database schema

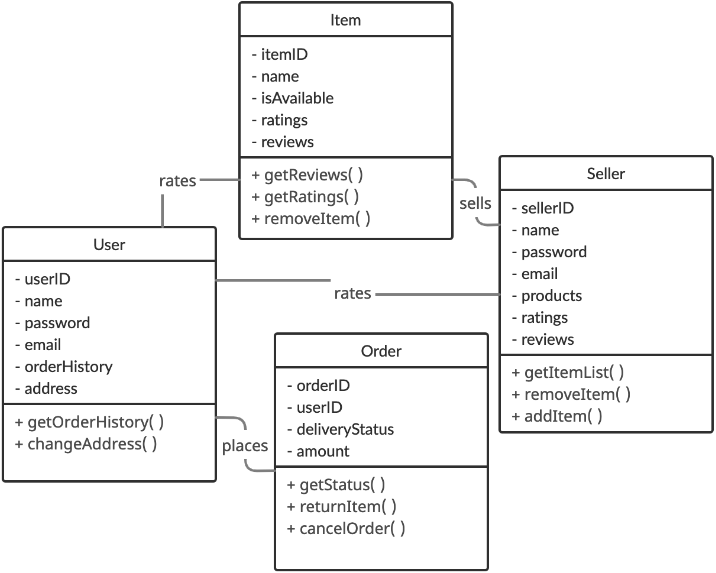
Buyer	
Buyer ID(PK)	int
Name	varchar(256)
Password	varchar(20)
Email	varchar(256)
Order History	varchar(1024)
Address	varchar(512)

Seller	
Seller ID(PK)	int
Name	varchar(256)
Email	varchar(256)
Password	varchar(20)
Products	varchar(1024)
Ratings	float
Reviews	varchar(256)
no. of Orders	int

Item	
Item ID(PK)	int
Name	varchar(256)
IsAvailable	bool
Ratings	float
Reviews	varchar(256)

Order	
Order ID(PK)	int
Buyer ID	varchar(256)
Delivery Status	bool
Amount	float

Step 7. Class diagram



Step 8. Algorithm

WAREHOUSESERVICE($User$) ▷ Manage inventory

Input: Item ID, Available items, order ID
Output: Maintain inventory as the items are selling

1. Maintain indices of all the products available in all warehouses
2. Decrement item count as the item sells
3. Increment item count if returned

ORDERPLACEMENTSERVICE($User$) ▷ Order Placing

Input: Item ID, Available items, order ID
Output: Maintain data for order

1. Record orders by buyer in relational database- MySQL
2. Reflect the change to inventory database as soon as the item is ordered

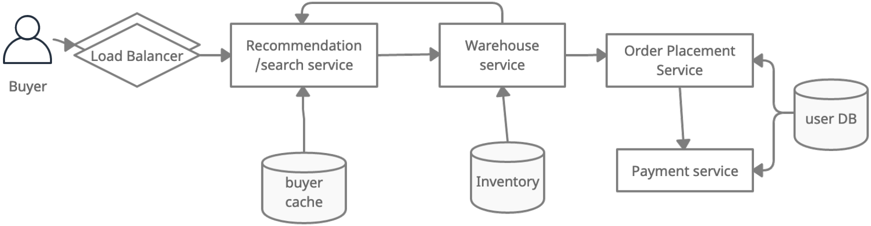
Step 8. Algorithm

RecommendationService($User$)	▷ Recommend products

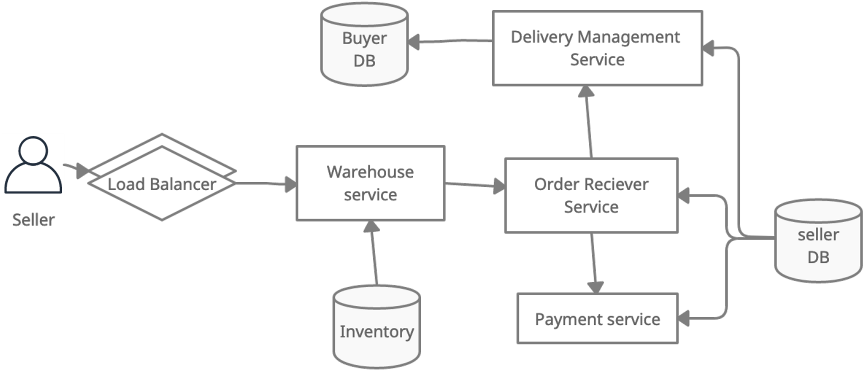
Input: orderHistory, sellerID, ratings
Output: Recommend related items to the buyer

1. Normalize other buyer ratings for a particular product
2. Perform user-user collaborative filtering to recommend products
3. Perform item-item collaborative filtering to recommend more products
4. Recommend other items sold by the same seller the buyer has shopped with before

Step 9. Service diagram



Step 9. Service diagram



Step 10. Business model

- Commission-based model

References

- E-commerce website system design
- Amazon System Design by codeKarle