Image Gallery App

↳ Reflective.gpad

Features

- Upload images
- View images
- Download images
- Tag images
- Share albums (or tags?)
- Bulk actions
- Memories to rediscover images?
- prescription on images
- Search / filter images

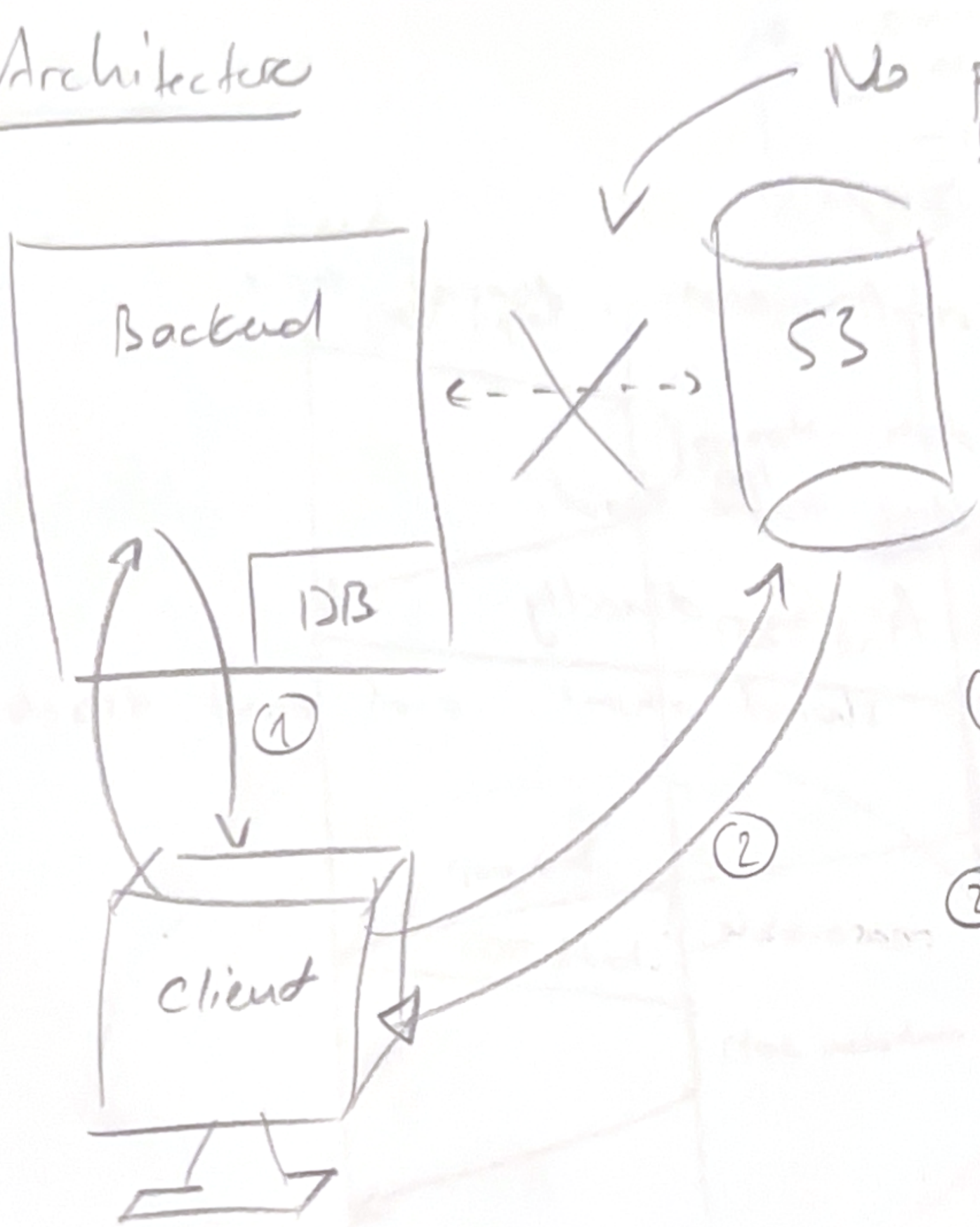
MVP I Scope

- Upload
- View
- ~~Add only magic link~~

MVP II Scope

- Add only magic link
- tag system
- Download images

Architecture



① Get presigned URL from BE

② Fetch and upload image data from/to S3

Issues

- Request delayed by extra roundtrip time
- Backend metadata possibly not in sync with S3
  → reconciliation?

Advantages

- Backend can scale further
- Client can get local S3 provider for faster download
- Client needs second vendor setup
- Backend only stores metadata, small DB size
- Backend can still control how much data is consumed by users

Alternatives

- Backend stores images
  → possible for small operation, doesn't scale!
- Backend proxies data
  → Backend has large traffic load!
- Frontend has keys for S3 directly
  → doesn't work for stored albums, and can't track traffic!

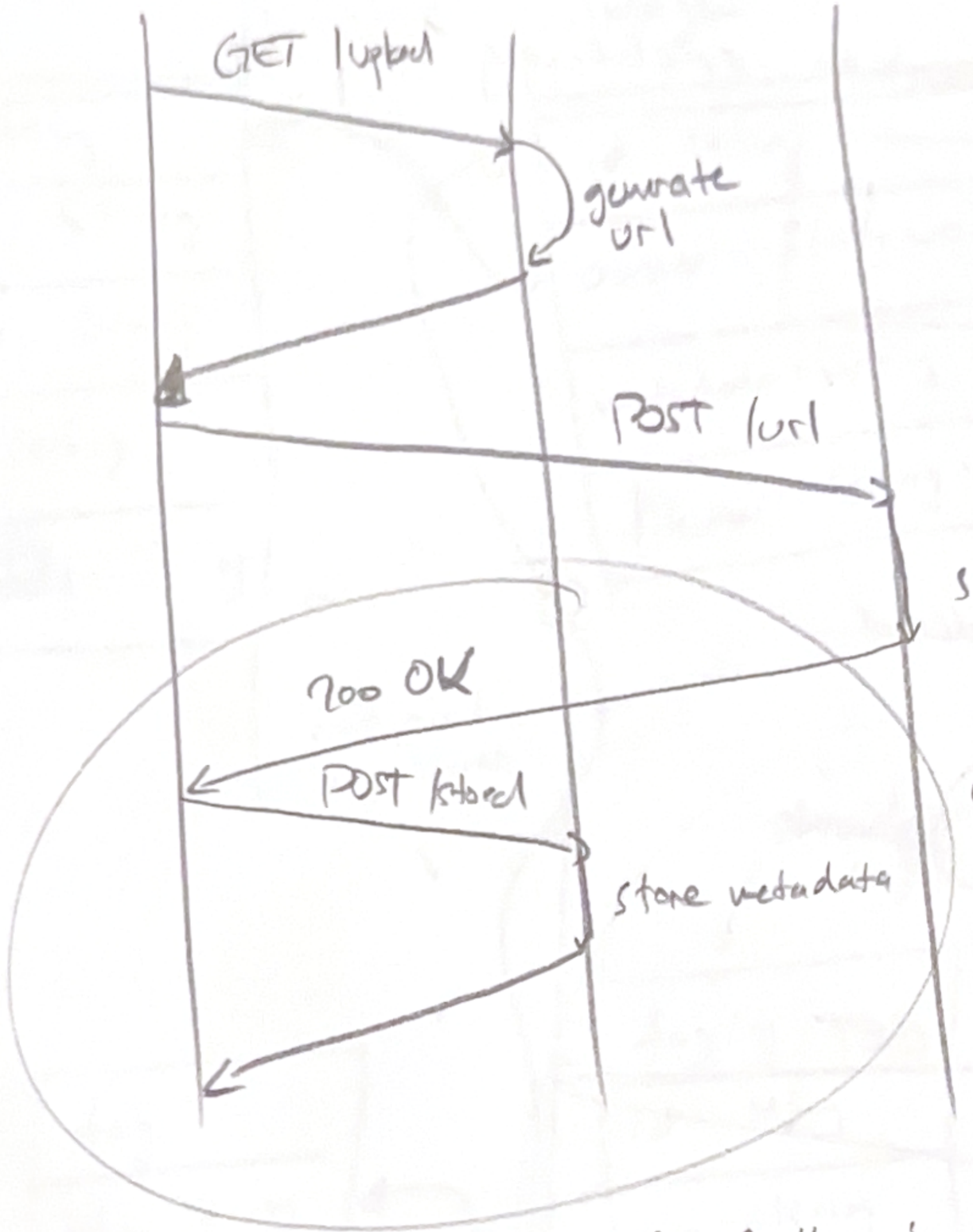
→ Approach seems reasonable!

Flows

Login:

Upload image:

VO communication between BE and S3
→ possibly on airgapped network!
→ good stuff



→ Client must persist state :)

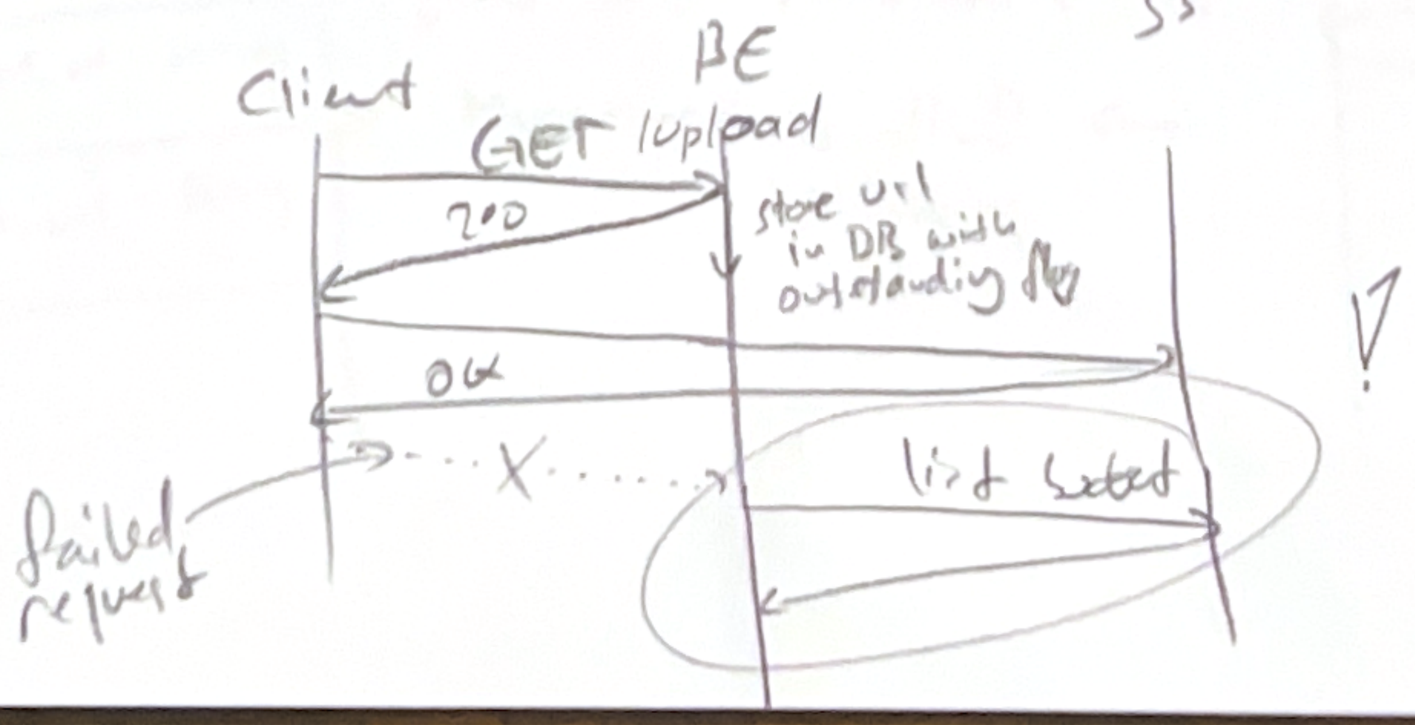
crucial that BE is informed! Otherwise there is orphaned data in S3. → incurs cost to the user

→ Approach 1: Eventual cleanup

Let that happen but occasionally list bucket and delete orphaned data / surface mismatch to user

→ Approach 2: Immediate reconciliation

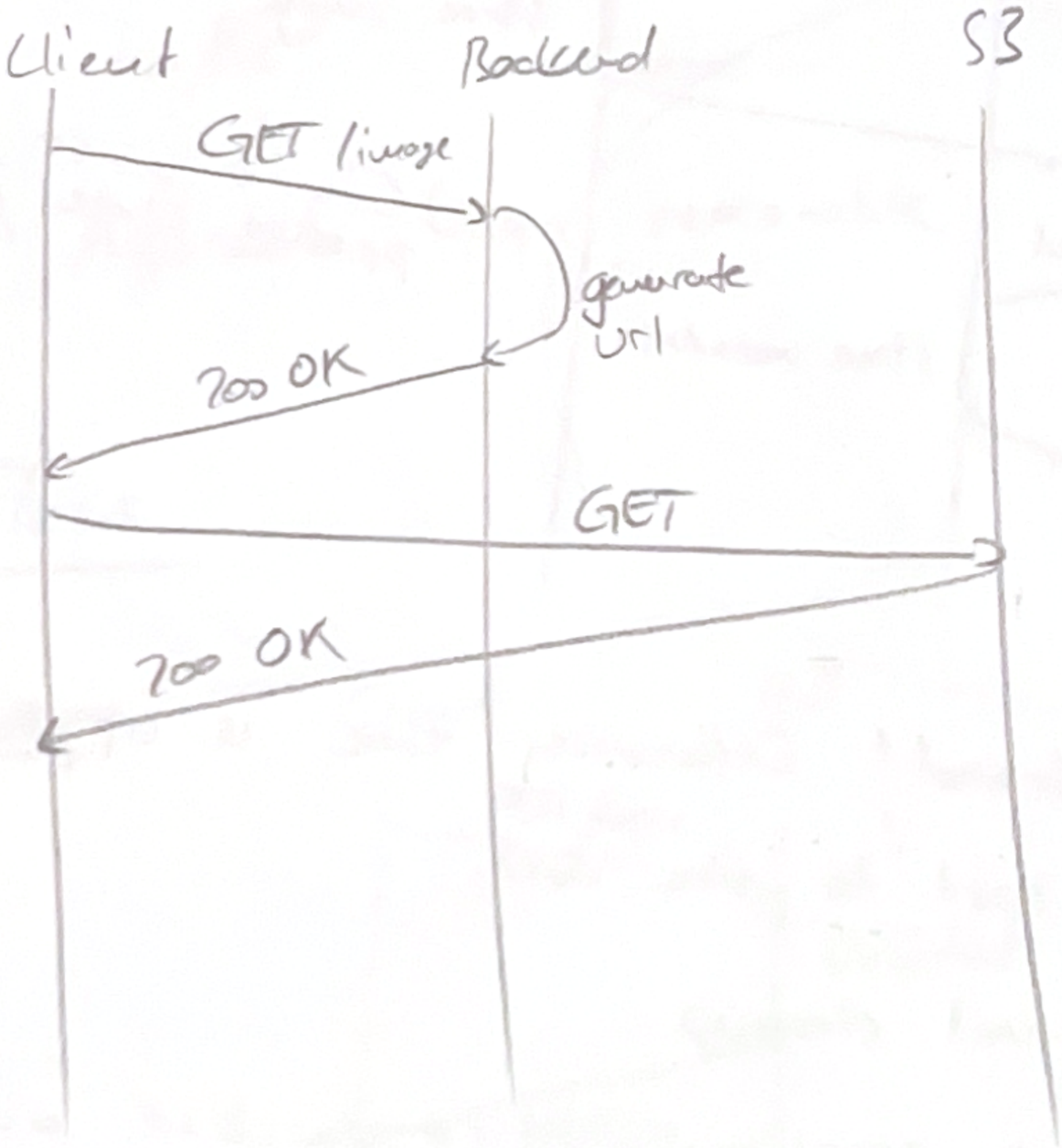
Handshake with BE:



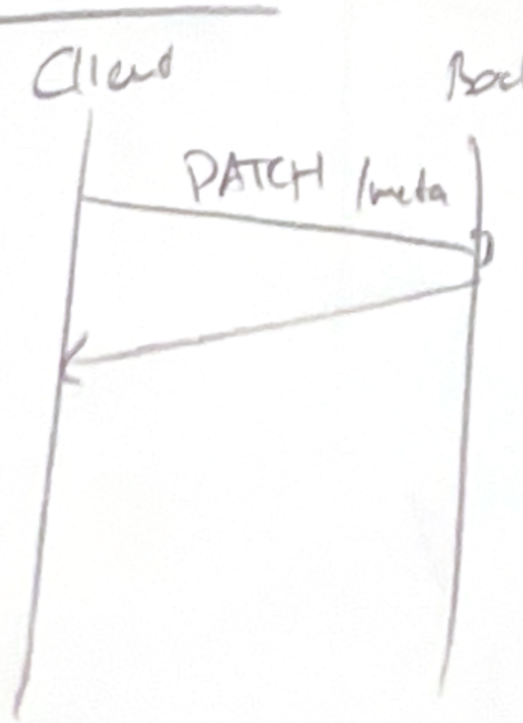
→ This introduces communication between BE and S3!
  → possibly S3 is self-hosted and not accessible from BE
→ Should BE even wait for report from FE or just use that mechanism exclusively?

→ Major Question!

Download image:

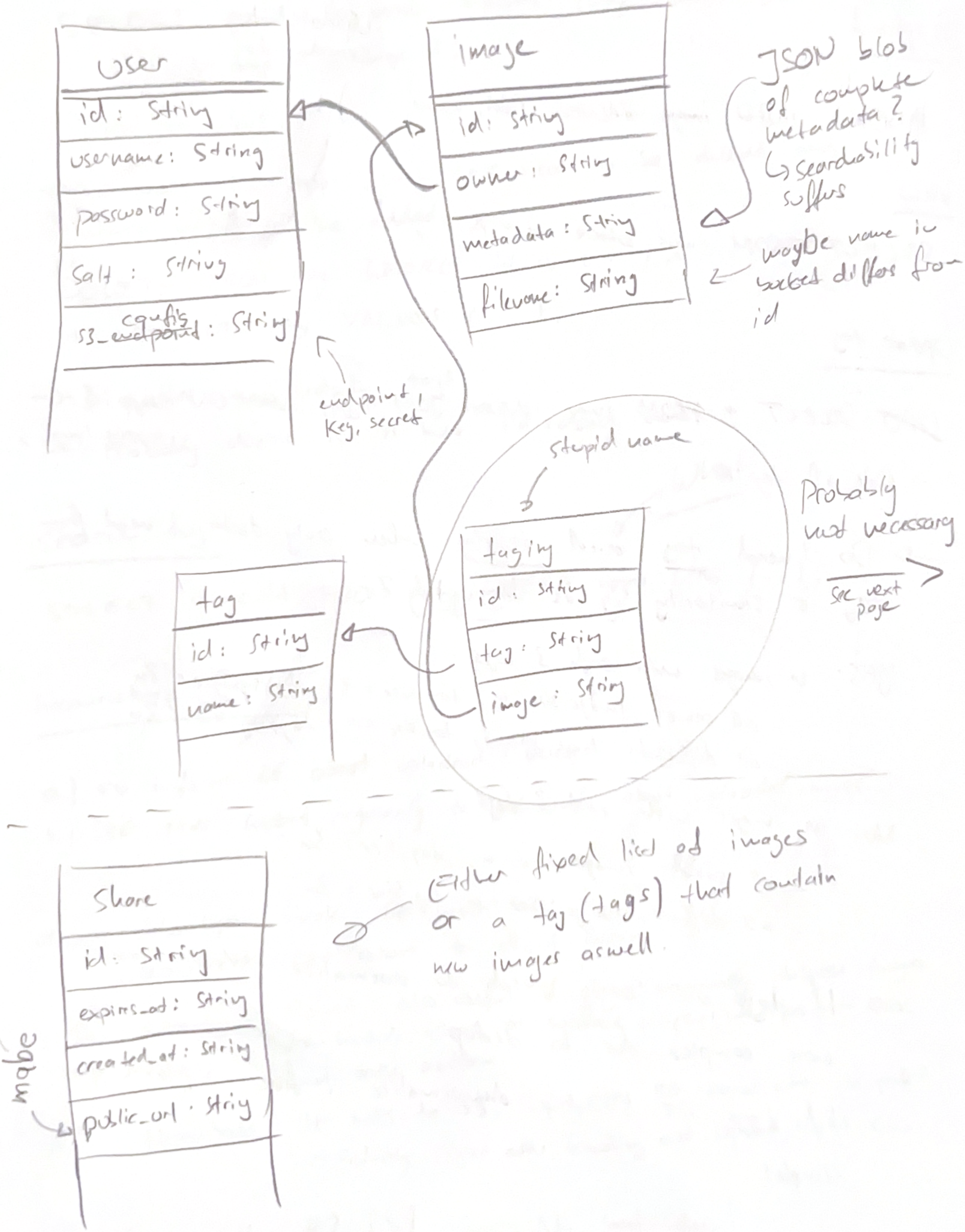


Change Tag:



→ Individual file only?
→ Bulk action must be possible!

DB Schema


user
id: String
username: String
password: String
salt: String
s3_~~endpoint~~ configs: String
endpoint, key, secret
image
id: string
owner: String
metadata: String
filename: String
JSON blob of complete metadata?
↳ searchability suffers
maybe name in bucket differs from id
stupid name
taging
id: String
tag: String
image: String
Probably not necessary
see next page
tag
id: String
name: String
share
id: String
expires_at: String
created_at: String
public_url: String
mabe
Either fixed list of images or a tag (tags) that contain new images aswell.

Queries

Upload image
View image
Update tag
search tag

Upload

INSERT INTO image VALUES (vid, owner, ...);

View

SELECT * FROM image where id = X ;

Update tag

~~UPD~~ SELECT * FROM image INNER JOIN tagging ~~LHER~~ ON image.id = tagging.image ;

→ List of tag ids

→ Do I need tag and tagging when only data I need from tag is similarily sig as tagging.tag ?

YES: n images with each 3 tags
- all same: Image.size = n, tag.size = 3, tagging.size = 3n
- all different: Image size = n, tag.size = n, tagging.size = 3n

No: n images with each 3 tags
- all same: Image.size = n, tag.size = 3n
- all different: Image size = n, tag.size = 3n

→ If tags are long, need to normalize. → Makes queries more complex due to dedup.

→ If tags are short, denormalize is fine → Makes queries simpler

↷ update tags

SELECT tag.id, tag.name FROM image INNER JOIN tag
    ON tag.image-id = image.id;

→ list of tags

→ compute in App which tags need to be deleted and which need to be added.

DELETE FROM tag WHERE id IN (...);

INSERT INTO tag VALUES (...);

→ 3 queries to update tags

→ Probably doable in ∧ but I'd rather not

search tag

What?

SELECT image-id FROM tag WHERE name LIKE (...);

Inconsistency Revisit

a.) FE informs BE about uploaded / deleted images.

b.) BE fetches backet regularly to get info about uploaded images.

a: Advantages: Simple, BE never talks to S3

Drawbacks: Not robust to client failure

b: Advantages: Robust, also useful if client manually deletes images
drawbacks, allows to surface inconsistencies to user
and take action

Drawbacks: BE talks to S3, how does BE know when to poll?
→ Probably right after handling past upload lives.

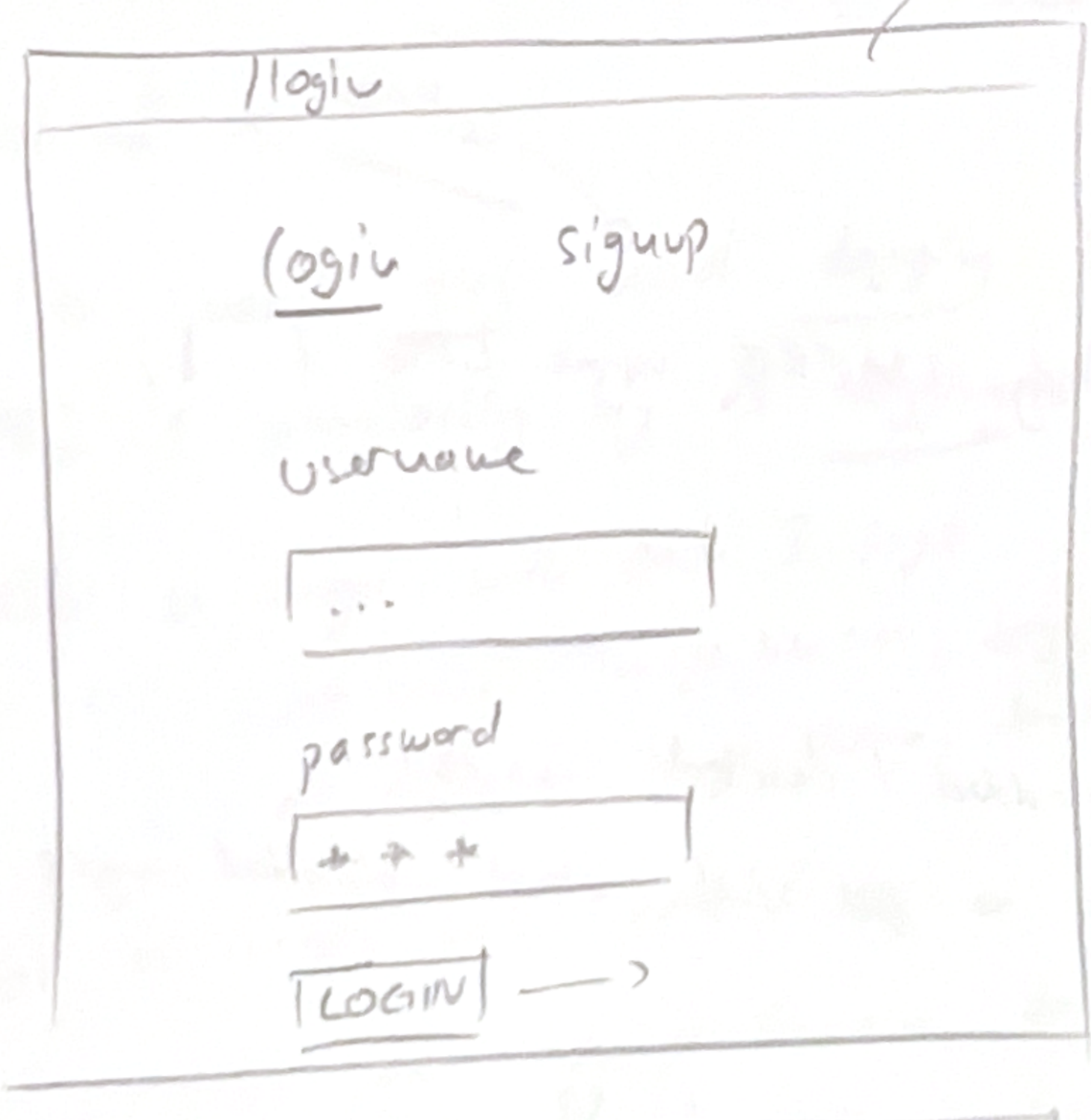
→ Go with #2 (b) → No, bad idea!

UI Concept

Screens for MVP:

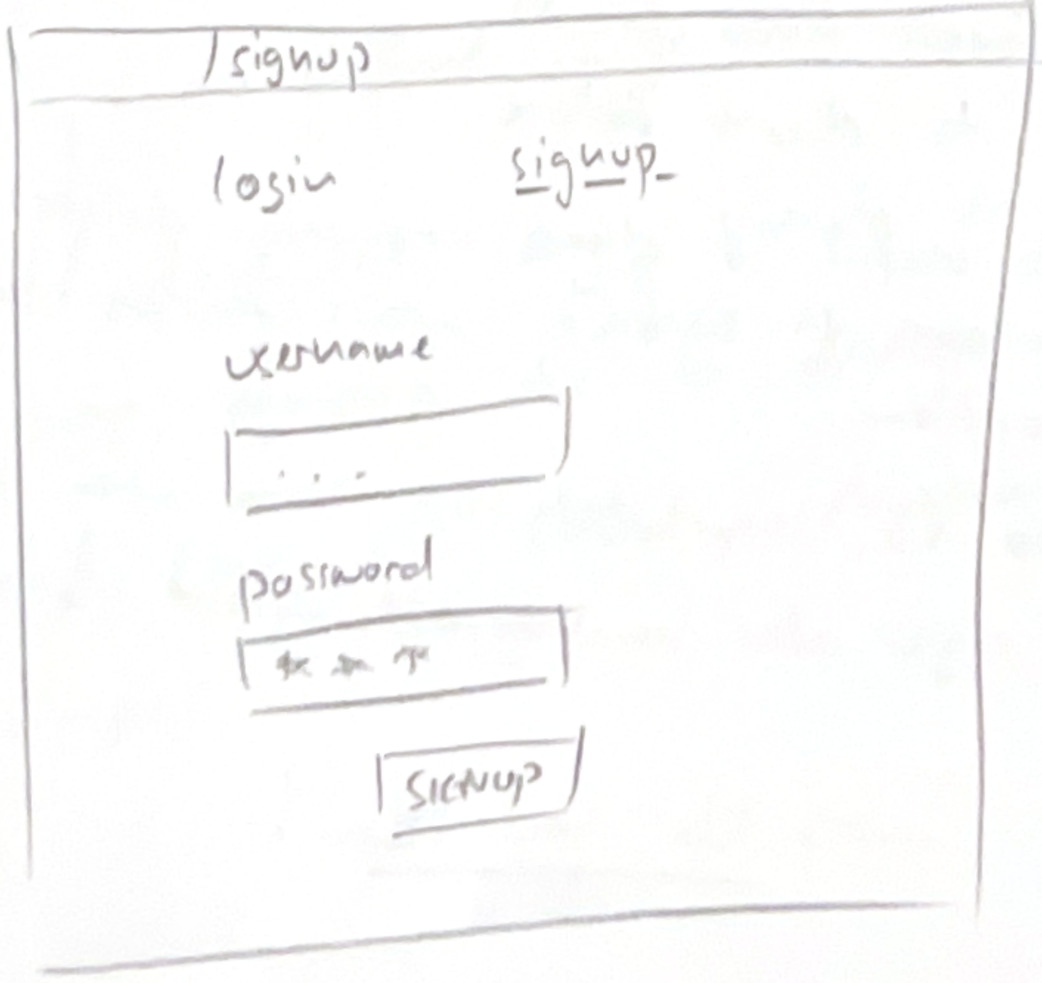
- Login → Simple username / pw for now
- View → Style column only for now (no lightbox)
- Upload → Bulk upload / downscale

Login



→ persist screens in URL from now on!

→ Simpler design than fieldnotes



→ No need for second password field, people use pw-manager anyway

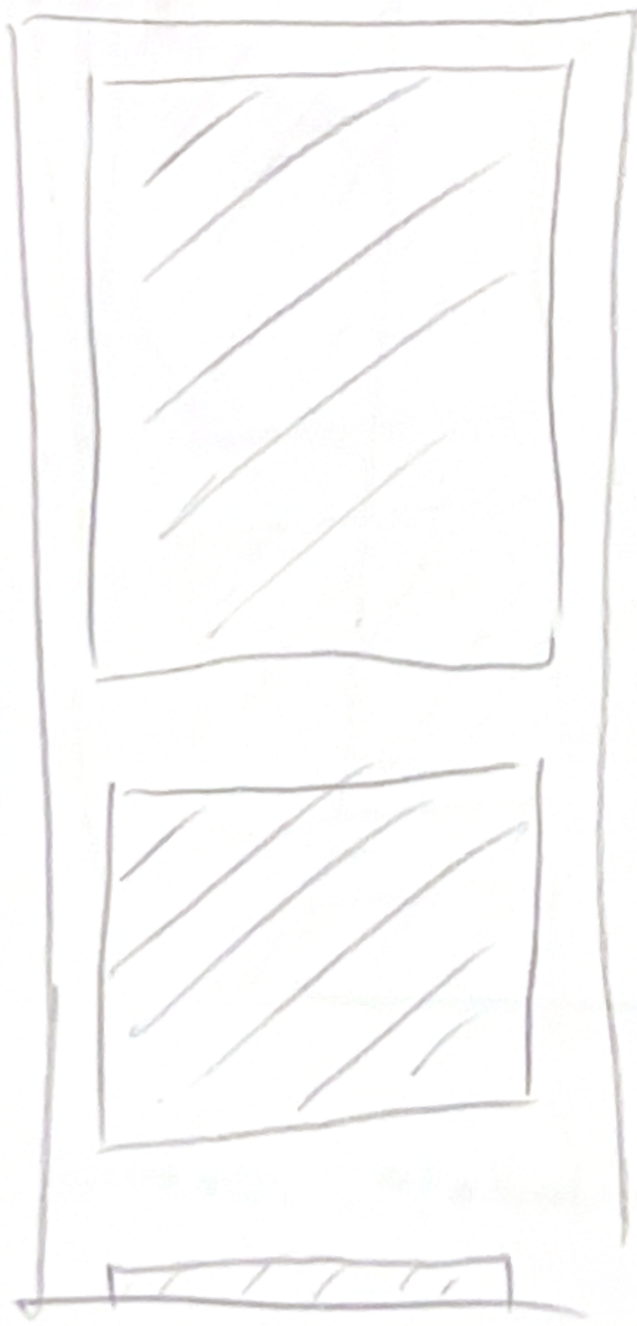
View

→ No icons anywhere!
→ every other screen accessed via shortcut

→ max whitespace

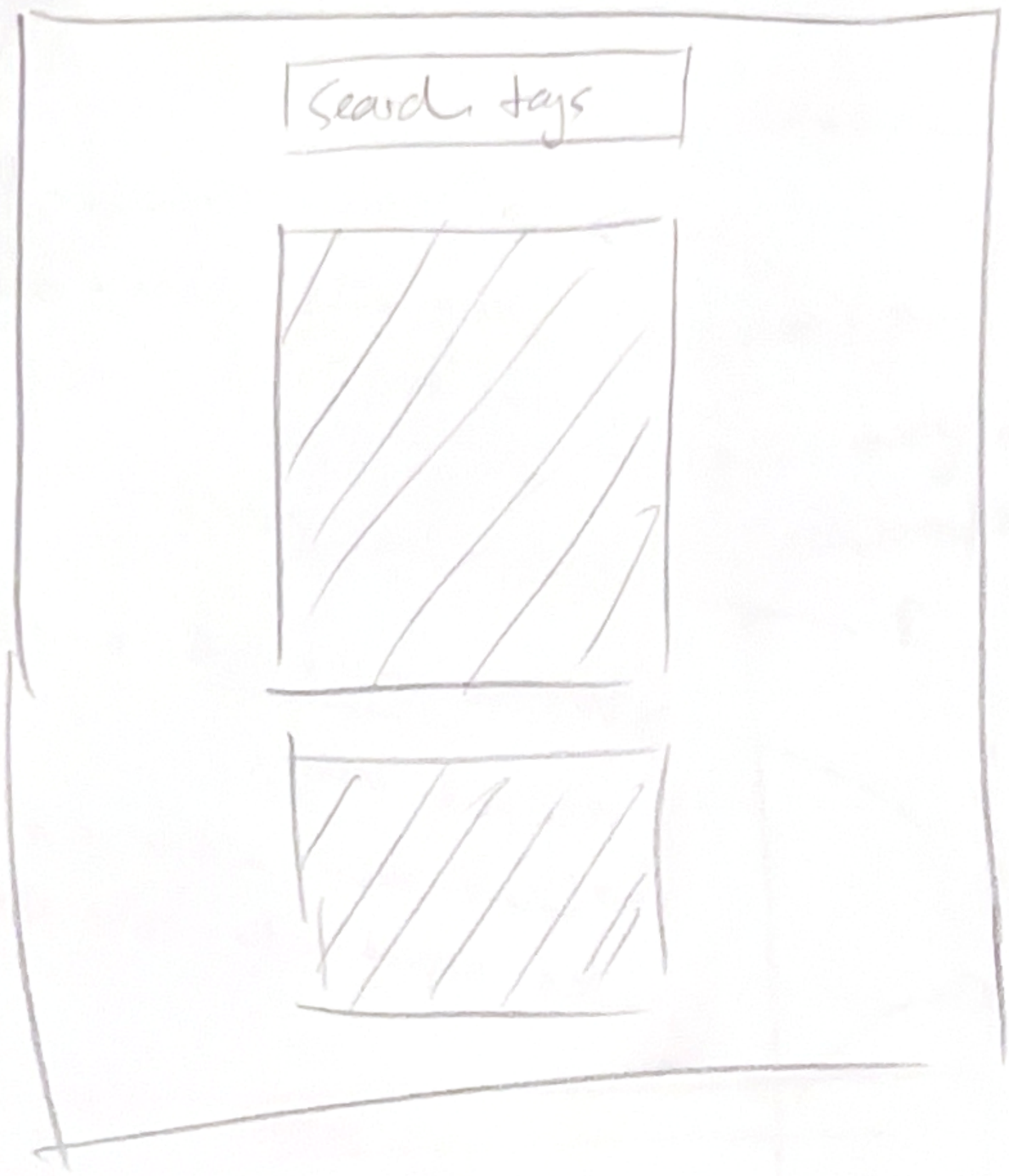
→ height must not exceed viewport height!
→ Limit max-width and let aspect-ratio take care of it

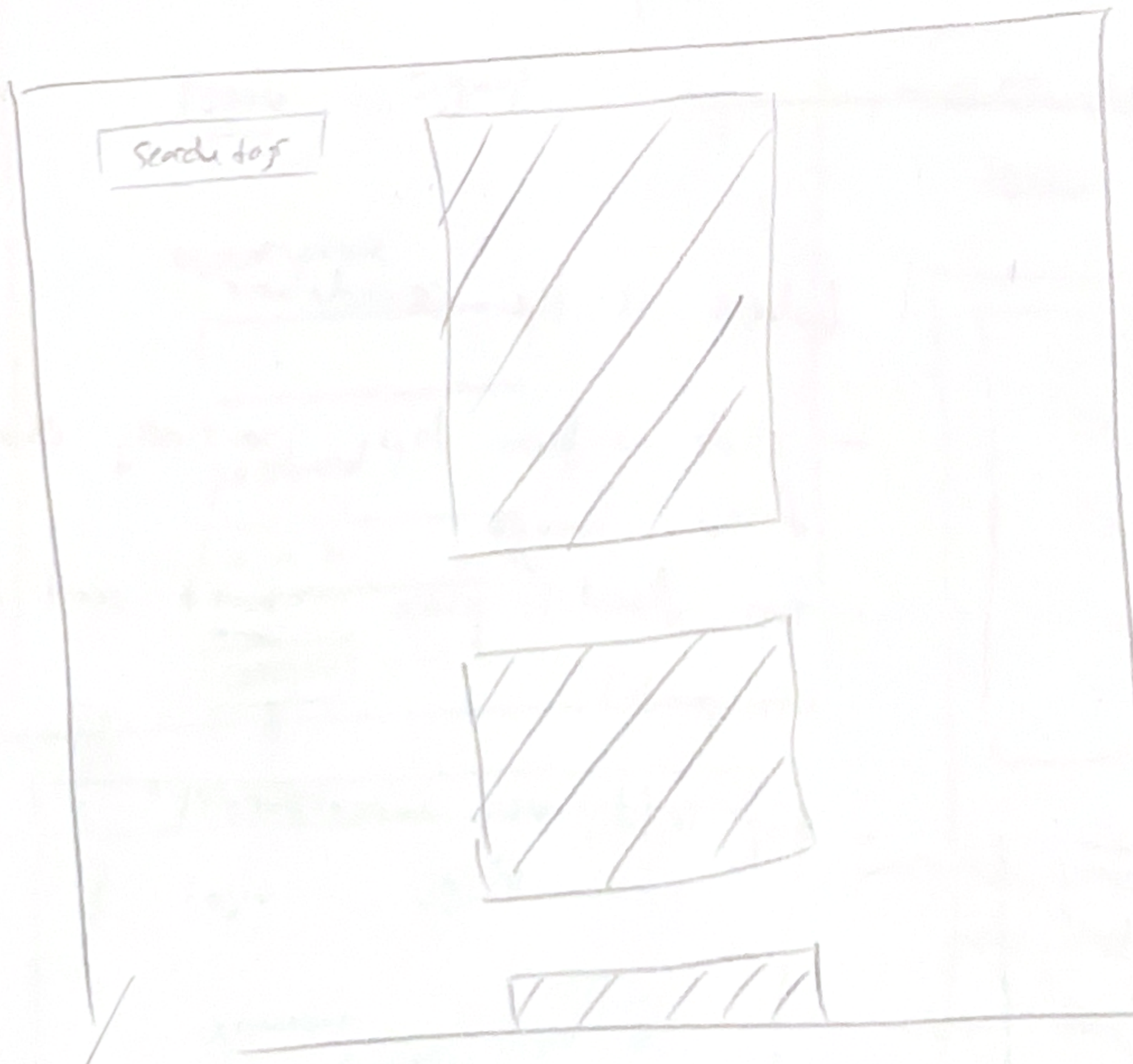
mobile



Where is the searchbar?

- slide in from top, partially obscure first image
- top, fixed in place → must scroll out of viewport!
- top left, very inconspicuous





→ Maybe more space for metadata, actions and so on?

Japan Chirigu

23 matches

download

delete

2023-7-20 -
2023-8-18

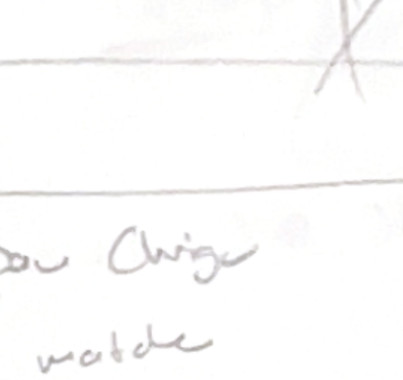
select all

a

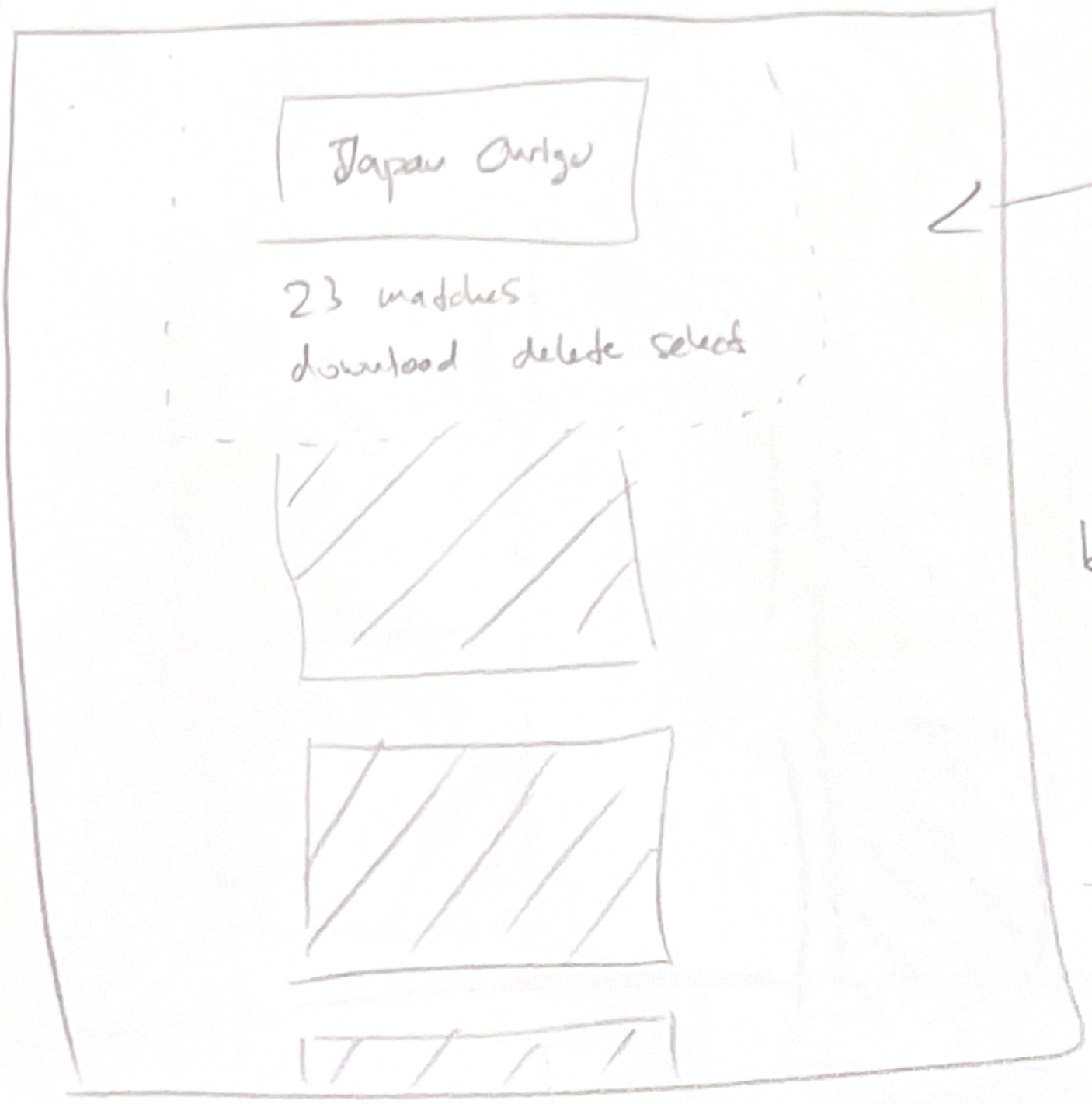
Japan Chiriga

23 matche

...



→ Show it on top in mobile view



↳ a feels more persistent, info is always in view
but can still be scrolled out of view immediately.

Selection

Allow shortcuts to rate and tag images quickly.
to add tags to multiple images, selection mechanism should exist.
→ what if a shortcut can be quickly configured to add
custom tag? → No need for selections
→ What if I want to download a selection?
→ Add 'selected' tag or shortcut and go through images
→ Requires sliding through each image, instead of click with mouse
→ Skip for now

Upload

→ Separate screen? Overlay?

→ if separate screen, must remain loaded to upload when navigating away.

→ TBD file structure on S3

Big bold like title on wireframing.dev

← ~~Back~~ View

Upload

UPLOAD

Formats: JPG, JPEG, PNG

← ~~Back~~ View

UPLOAD

DSC00A.jpg processing

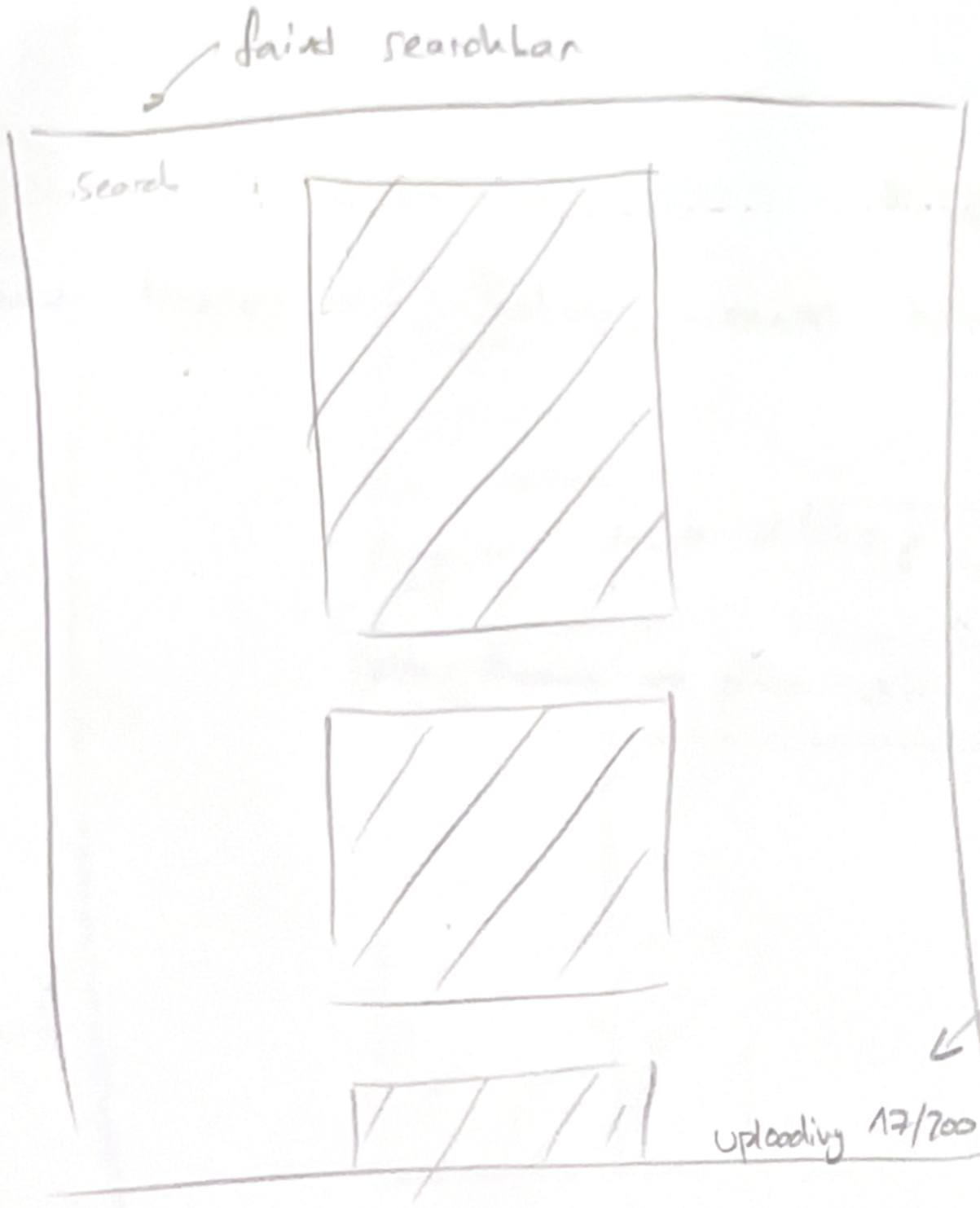
DSC002.jpg processing

DSC003.jpg uploading

⋮

DSC009.jpg done

⋮



File structure ou ??

FE preprocesses images for 3 levels : small, large, original

-> small is mobile and multi column desktop

-> large is single column desktop (and maybe fullscreen mobile)

-> original is for download

-> BE will identify images in bucket, so it must know which are versions of 1 image.

{id}-small, {id}-large, {id}-original ? X see there

-> How does BE know metadata about image ?

-> resolution, camera model, etc.

-> Option A: Don't care

Option B: FE saves metadata to separate file on S3

Option C: FE stores metadata as headers in S3 object metadata

Option D: FE asynchronously tells BE know later when retrieving image for the first time.

-> D is easy do retrofit later :)

-> BE defines URLs in specific structure:

GET upload_urls

{ [

{

small: String,

large: String,

original: String

},

{

:

},

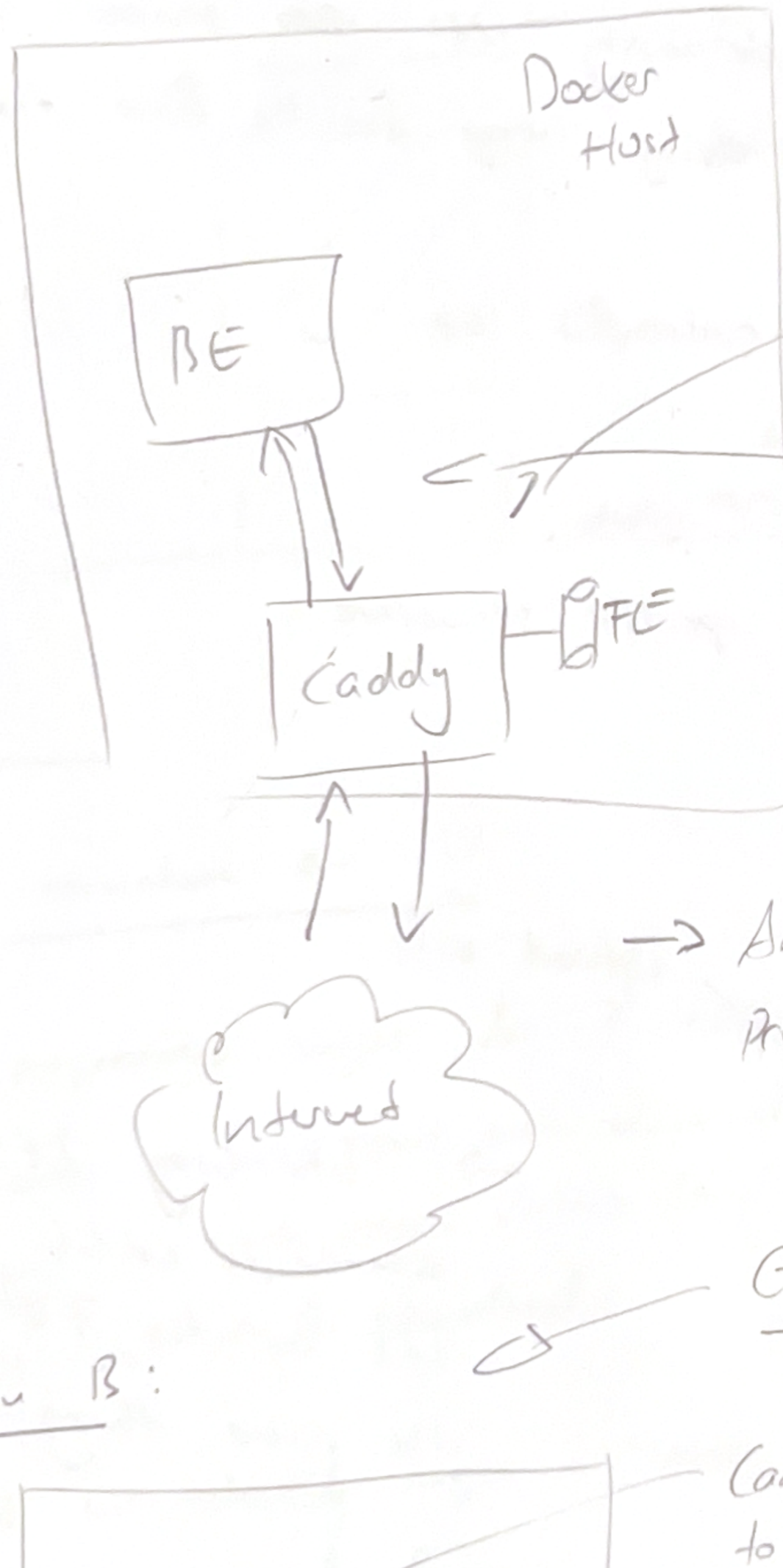
]

upload urls

-> This way BE knows which is which and it's up to FE to do the right thing.

Deployment

Option A:



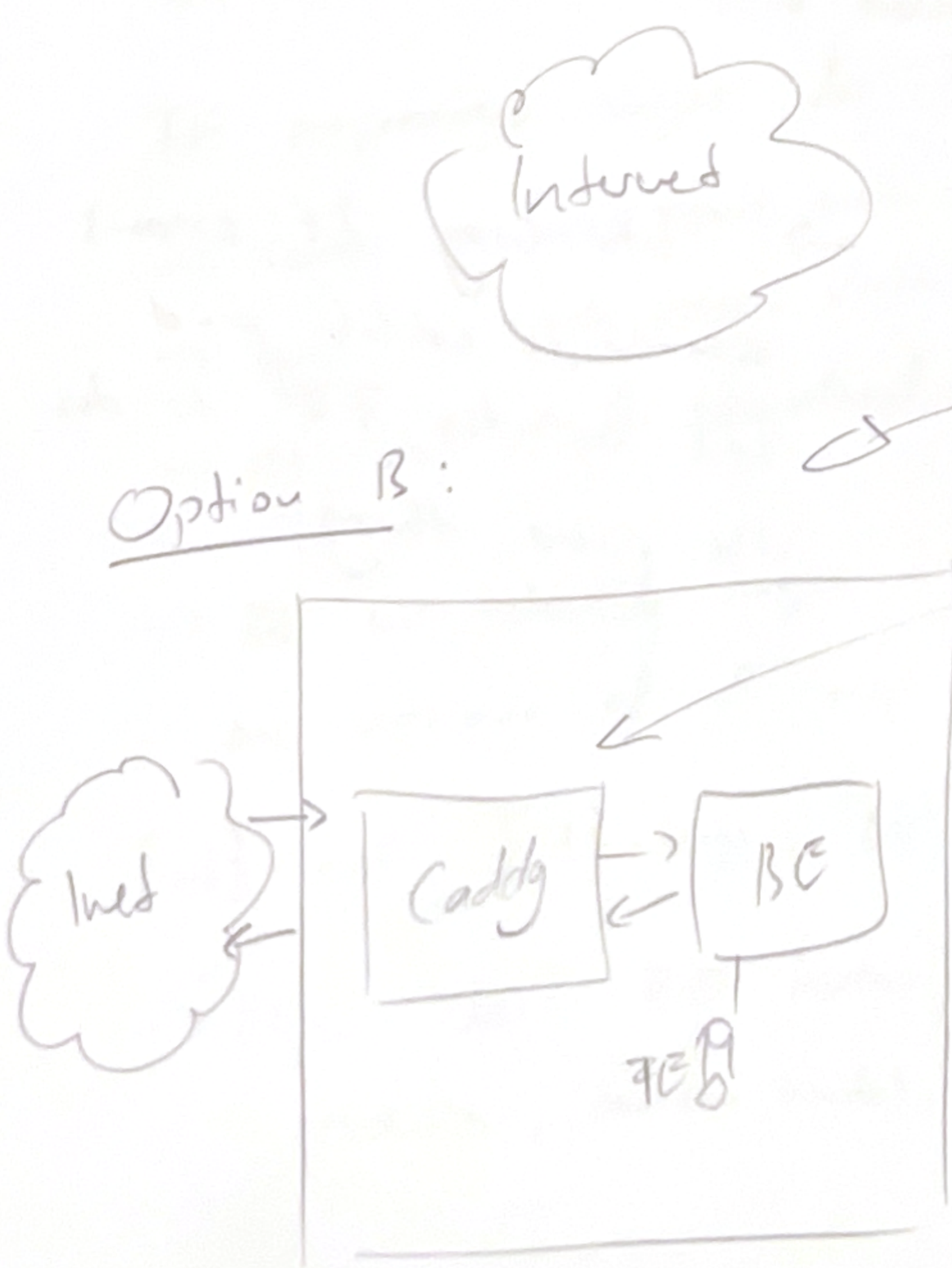
Freedom to choose subdomain vs subdir for api

→ Advantage: Simple, efficient

Problem: No metrics on FE
→ Caddy doesn't expose per-domain metrics

Go for this one.

Option B:



Caddy only does reverse-proxy to one domain

→ Probably FE BE would be served on subdir and not api subdir.

→ Actually, reverse-proxy two domains (x.dev, api.x.dev), in BE inspect host to serve api or static site

Malicious user attack vector

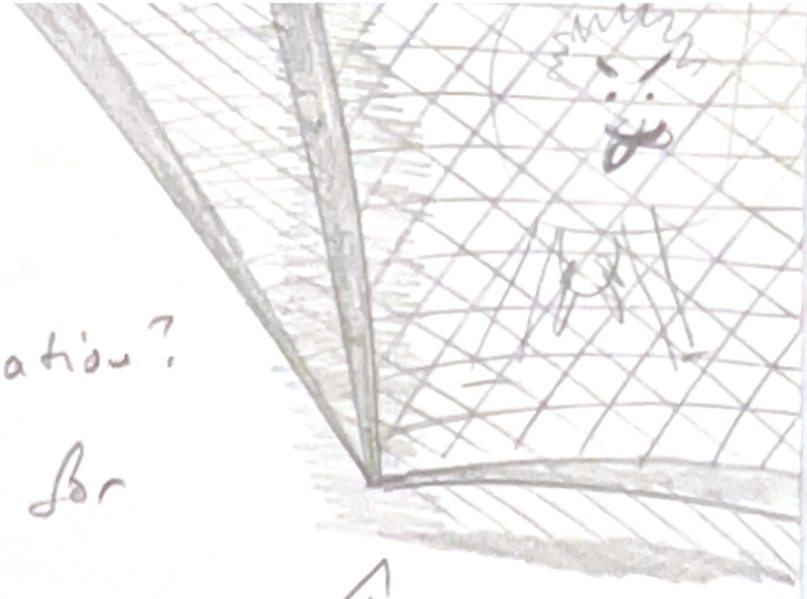
Is application intended for true multi-user situation?

→ Malicious user may attempt to cause cost for S3 like.

→ If only designed for self-hosting, no such considerations have to be made!

→ Do I ever need presigned get urls, or can I use the API key in the frontend.

Certainly for shared albums, I need presigned URLs



angry cow in prison

Features

- View images ✓
  - Single column "portfolio" view
  - Multi column "skim" view
- Upload images ✓
- Tag images ✓
  -> favorites, keywords etc are special case
- Create albums
  -> could be implemented as smart albums by tag
- Share albums
- Search & filter images by tag, rating or other metadata
- Automatic object detection and face recognition
- Download images in bulk
- "Memories" to rediscover images
- Freetext metadata on image to enrich shared albums with text.

MVP I Scope

- Upload
- View
- Simple auth
- Get DB schema right

MVP II Scope

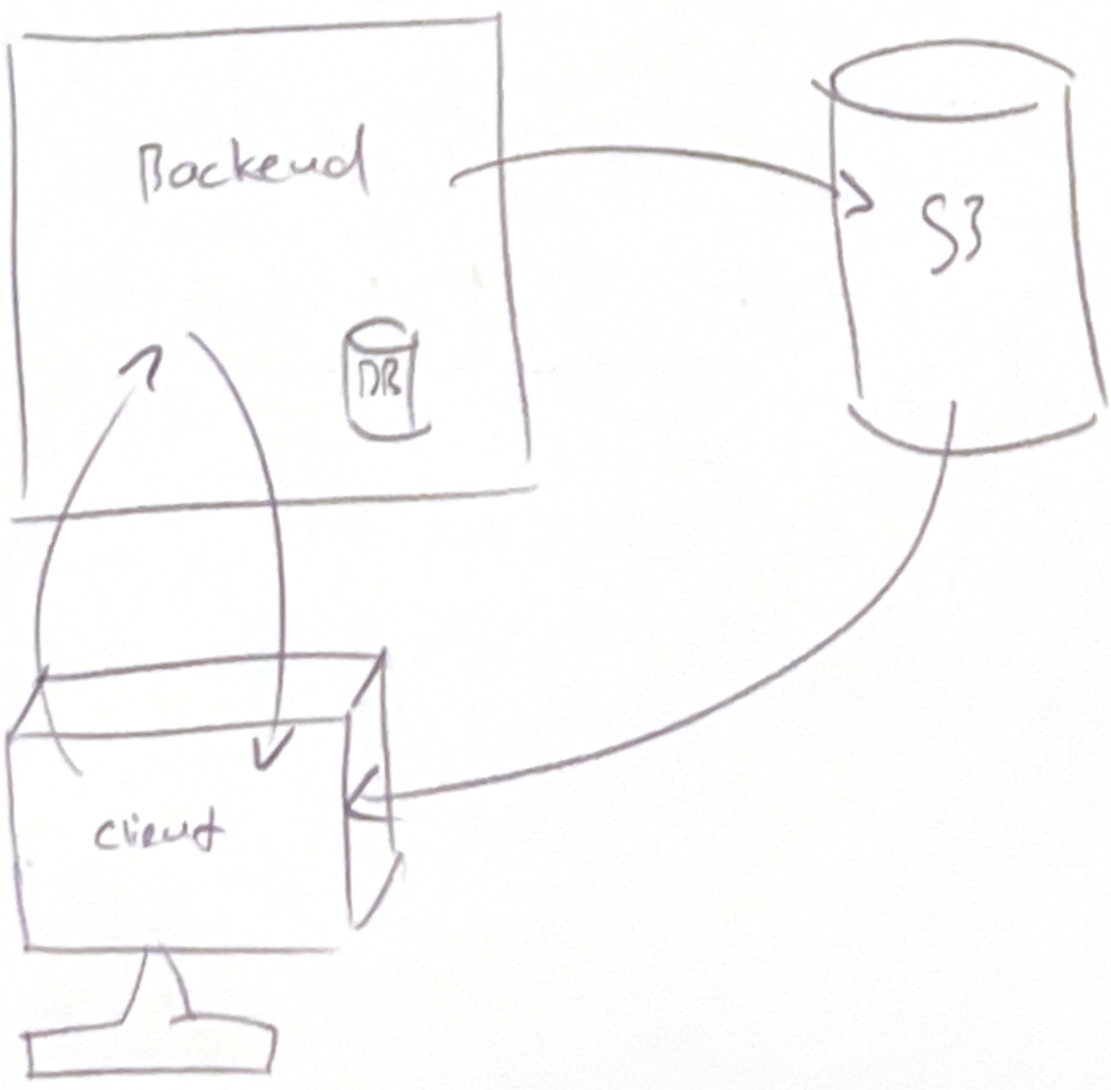
- Tag system
- Shared albums

Architecture

> Reminder to build for yourself and not overengineer for an imaginary userbase!

Constraints:

- I don't want to store image data myself
- I do want to store metadata in a fast SQLite DB
- I would prefer to avoid proxying image data through Backend
- A reasonable uprade path to E2E encryption is desirable but not a focus for now



- Client uploads to BE
- BE uploads to S3
- Client fetches from S3

-> BE still handles upload to do image compression and other tasks.
-> Client fetches via presigned URL from S3 directly to unburden BE.

(-> Can be compatible with E2EE later)

Classic Rust BE with SQLite DB and SolidJS FE should do the trick

Schema

Entities: User, Image, Tag?, Share?

Tags are conceptually a list of words that are attached to an image
Should they be modelled as a JSON blob on the image or as a separate entity that needs to be brought into relation with the image table?

denormalized approach

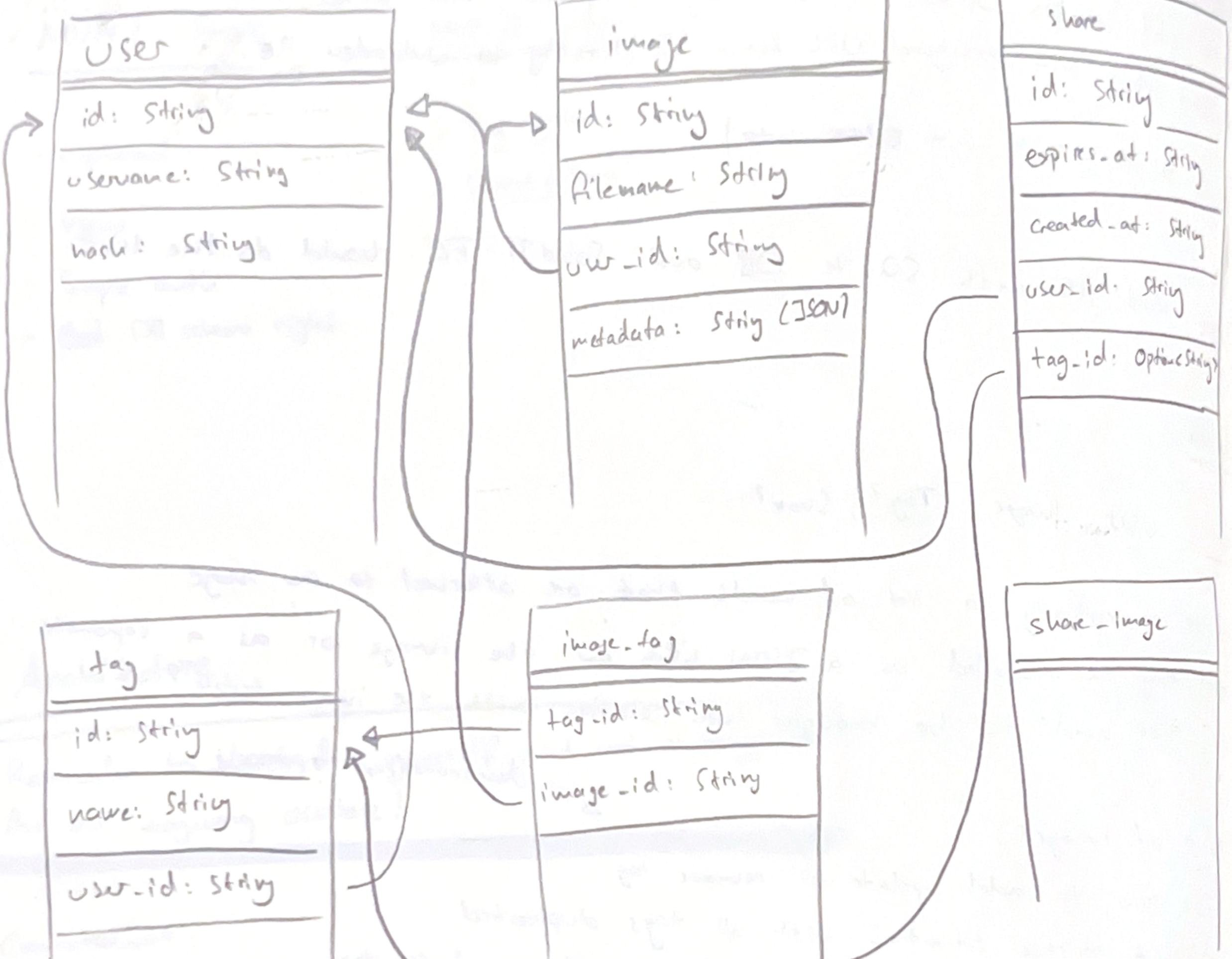
Attribute of image:

+ simple query to add, update or remove tag
- inefficient storage situation with all tags duplicated
- Bulk operation on multiple images' tags: very slow and complex
  -> need to read each tag attribute first to add, change, remove one and then write back. 2n queries for n images involved!

Tag entity with relation to image:

+ Efficient storage
+ Efficient search!
  -> Don't need to search n blobs, just + strings and then relate to the images
- Individual tag update now more complex
  -> add: check if tag exists, else create tag, create relation to image
  delete: remove relation, if no more usages of this tag, delete it
  change amounts to an add and remove operation.

-> Tag entity with relation to image probably nicer!



—> Do I need the user also in tag and maybe even image_tag?

-> Are tags shared across users?

—> Better to make tag user specific, the relation then probably doesn't have to care about user_id.

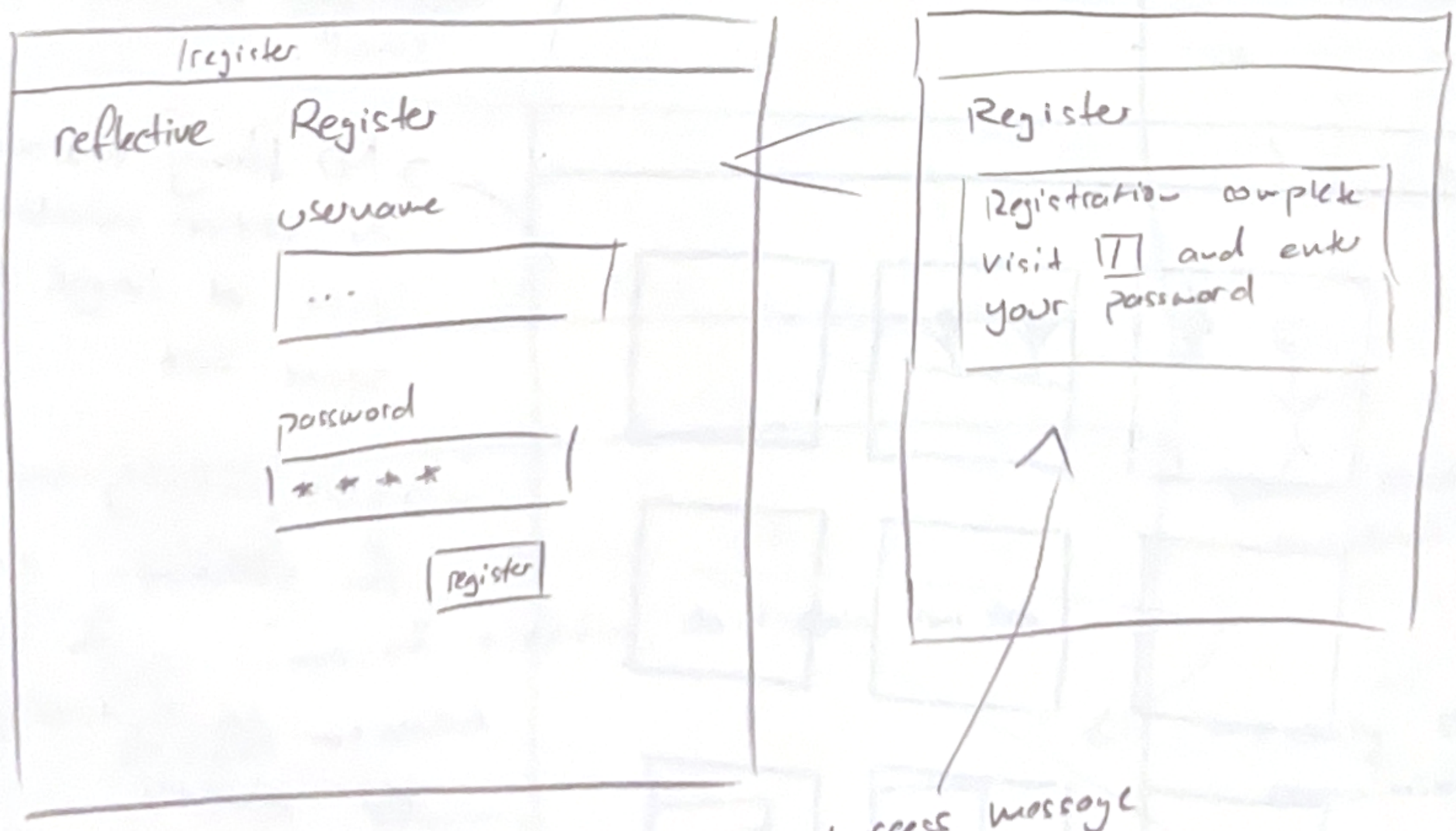
UI Concept

Basic routing idea: Always switch between screens via shortcuts
Don't use modal overlays, always use proper pages

Screens for MVP:

- Register
  -> Simple username/password input
- ~~Login~~
  -> not necessary, just use basic auth prompt for now
- View images
  -> single column, maybe multi column, no lightbox for now
- Upload images

Register



View

On desktop: No icons to simplify design

→ whitespace to not cramp layout

→ height of image will not exceed viewport height

↳ Not feasible due to portrait ing. Either portrait scroll across viewport or landscape be very small.

→ No fancy masonry grid or vertical swimlanes, just let image cover square box.

→ Will probably use this for skimming and then "zoom" in to single column view to actually look at pictures

order

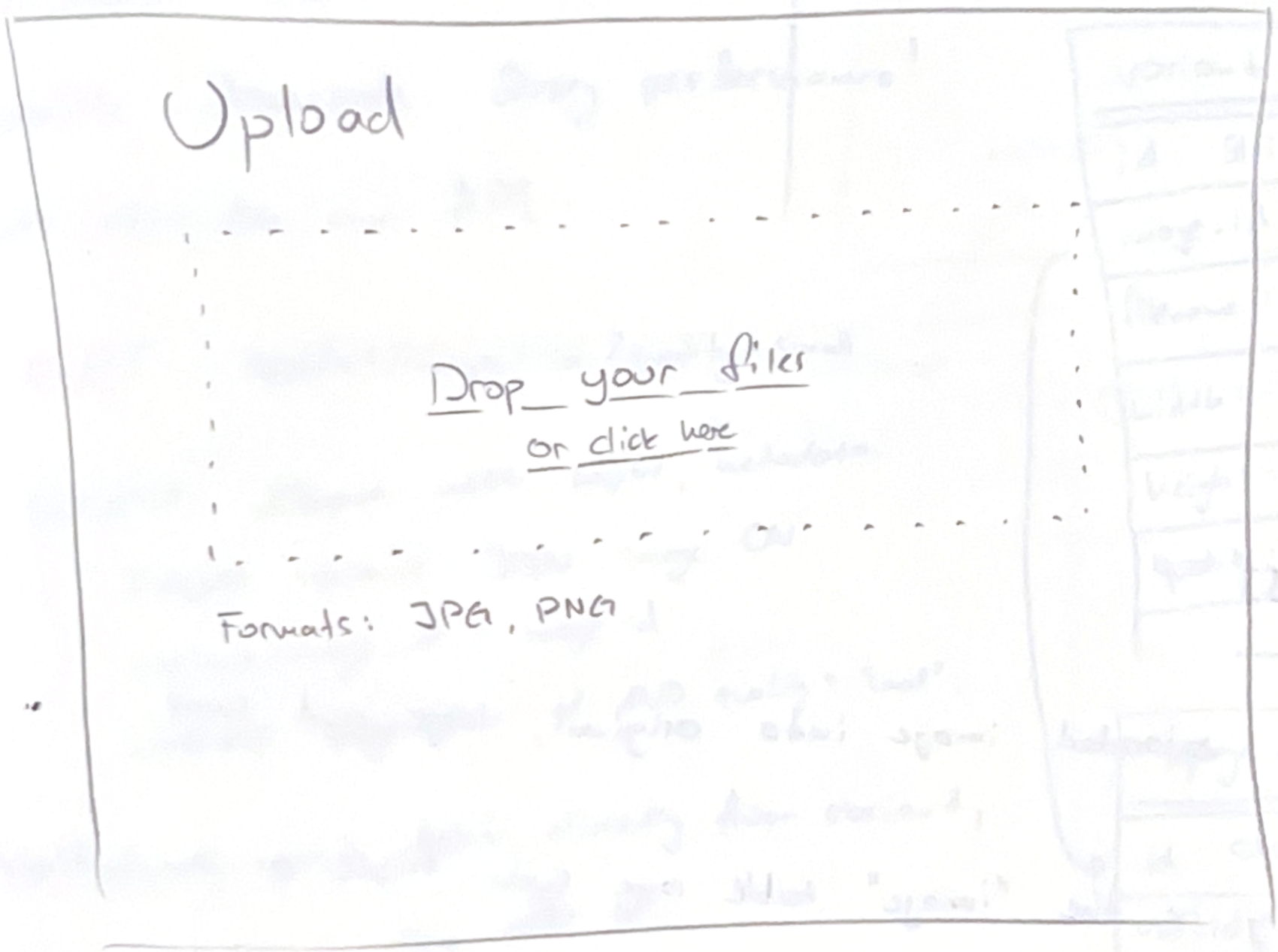
usability: must allow switching while retaining positio-

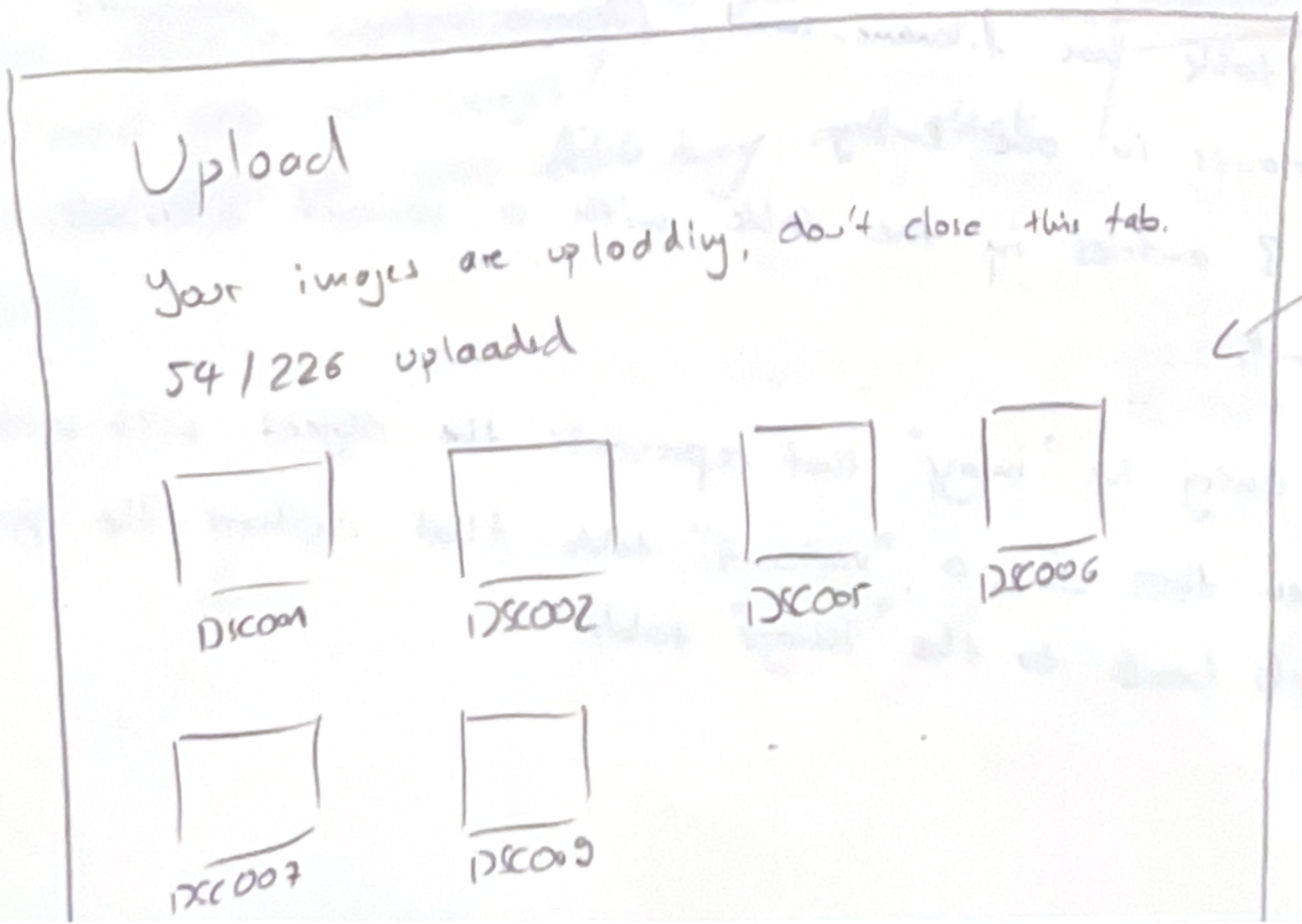
→ image-centric scrolling rather than document-height-based scroll positio-

Upload

Separate screen or overlay?

→ Separate screen for simplicity





show small image thumbnail if possible

→ only show remaining images

Upload

Your upload has finished.

visit your gallery now

↑
this is MVP I scope

File structure on S3

BE will process each uploaded image into original, large and small versions.

Question: Is an entry in the "image" table one such triple or an individual variant?

A → Either "image" table has filename_small, filename_large, ... attributes to capture all variants in one entry

B Or there are 3 entries in the table with a variant attribute indicating if its original, small, ...

C Or there is one entry in "image" that represents the object with metadata like toys etc but then there is a "variant" table that captures the processed variants that link back to the "image" table.

- Option B is not an option due to duplication of metadata, tags etc.
- Option A is acceptable but feels overkill due to the hardcoding of the variants in the attributes. (probably fine, really)
- Option C seems the nicest: "image" represents a user-facing domain object, thats where metadata is added, ratings are stored etc "variants" is an implementation-detail, thats where filenames on S3, compression levels etc. are of concern.

Possible downside: Query performance!

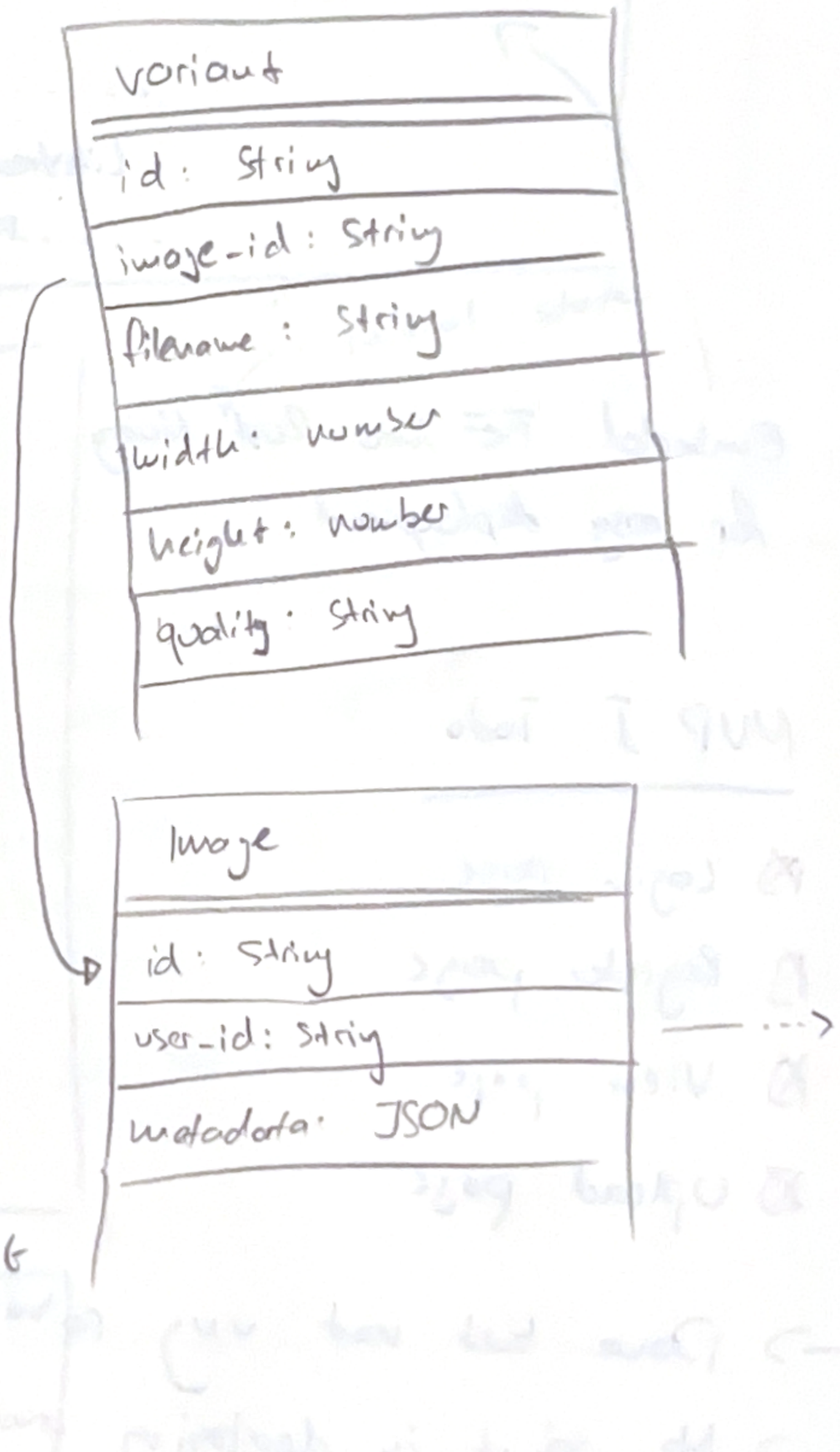
Lets consider an API:

GET /api/v1/images/:id:?quality=small

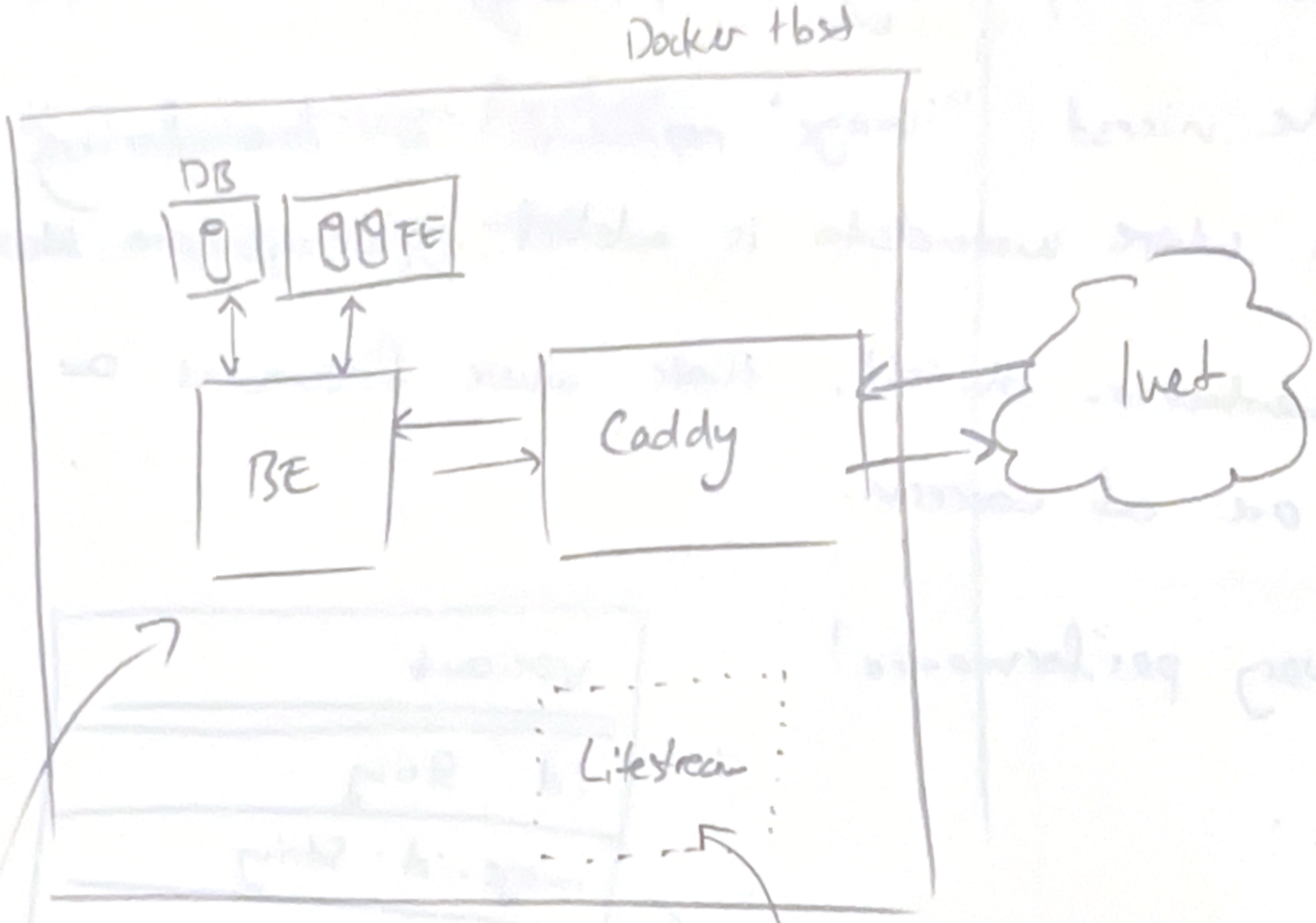
```
SELECT filename, width, height, metadata
    FROM variant JOIN image ON
    variant.image_id = image.id
    WHERE image.id = :id: AND quality = "small";
```

Or actually just fetch directly from variant, the image_id is there after all.

-> Does variant need user_id to ensure you can't read other users images?
-> Either this or also fetch image and check there



Deployment



Embedd FE into Rust binary for easy deployment

consider litestream to backup SQLite DB.

MVP I Todo

- [x] Login page
- [x] Register page
- [x] View page
- [x] Upload page

→ Done but not very robust or usable yet.

→ No point in deploying and using at the moment.

Things to do for MVP II: → This should make app deployable and usable!

- Get db schema right and future proof ✓
- Build simple tag system - build simple search
- Get image compression right
- Get S3 storage format right ✓

→ Make reflective personally usable!

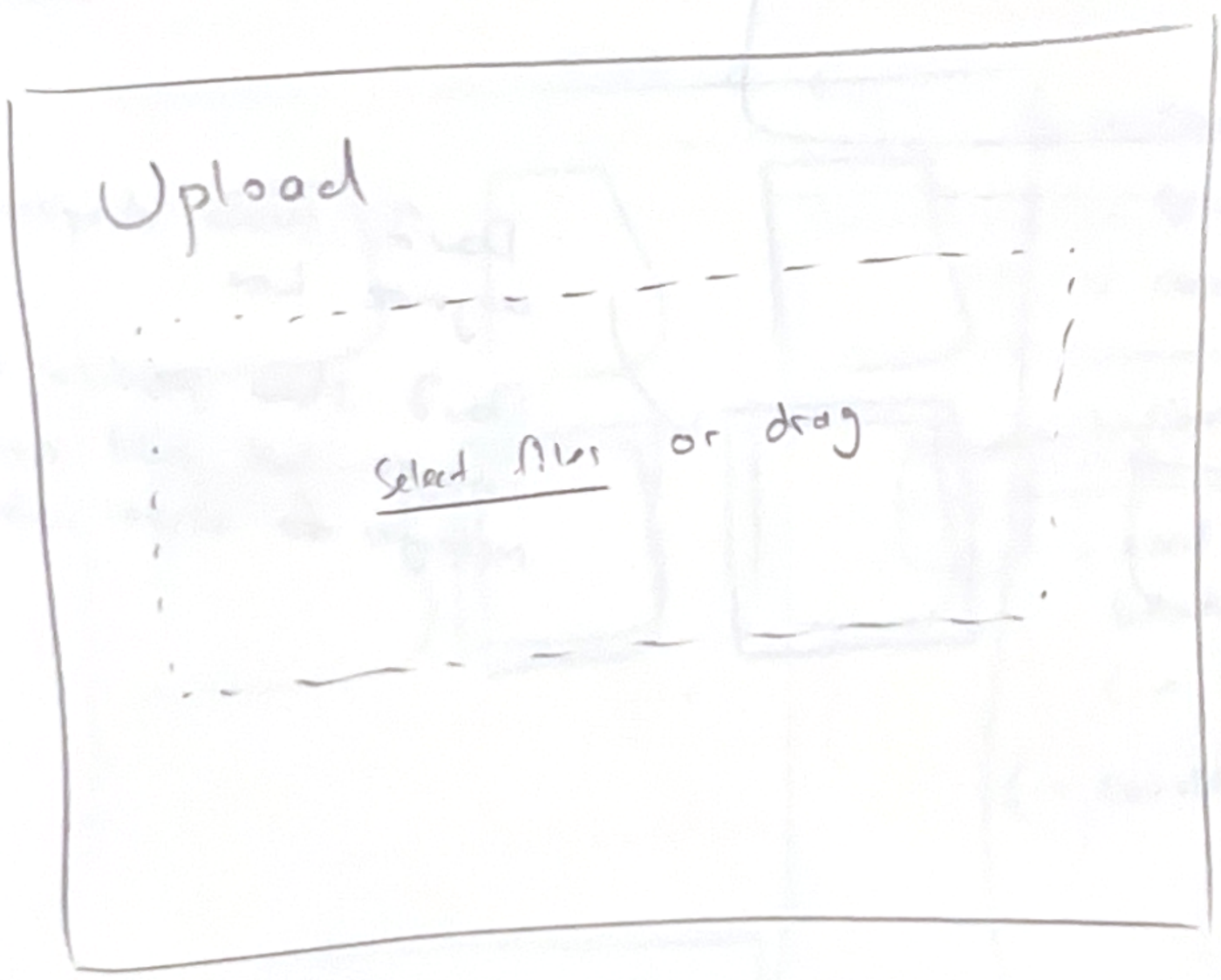
→ Goal for go-live

Random thought: By making compression etc. not happen at upload time, but async by downloading and processing from S3, iterating on compression is way simpler

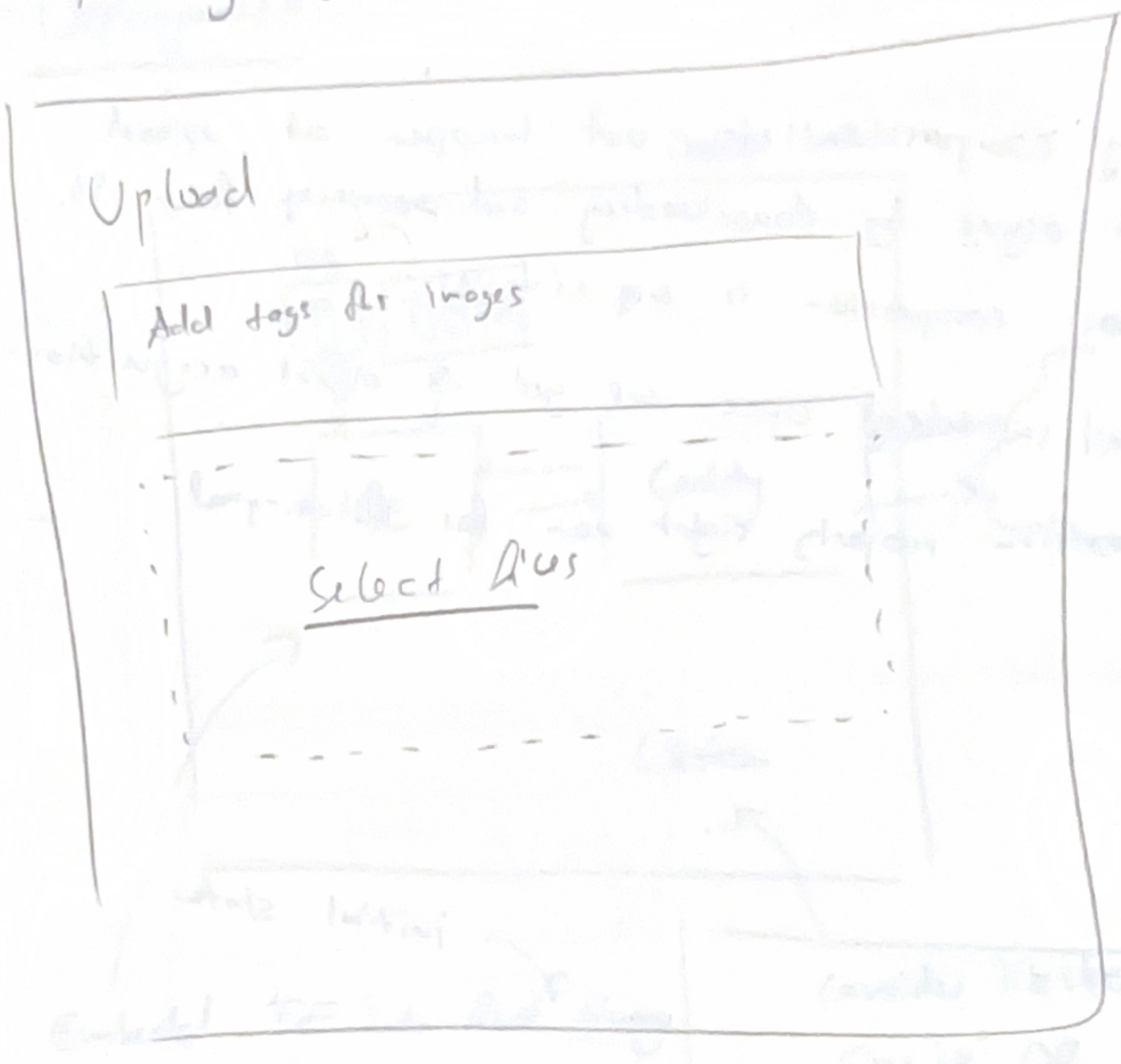
→ can be switched once we get to object recognition

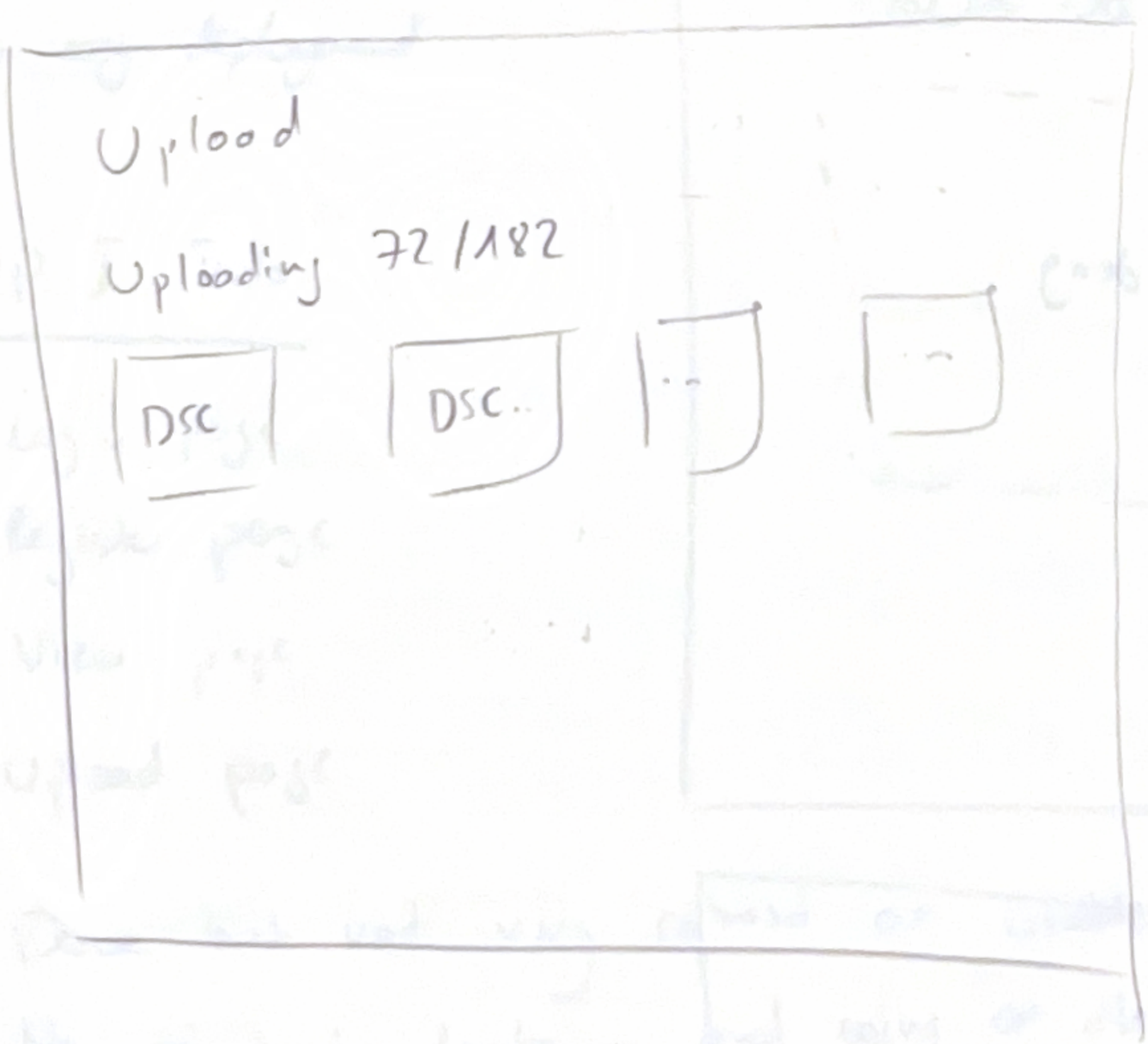
→ Adding version property right now for future-proof

Upload UI revisit



→ Allowing tags to be specified while uploading is nice but more complex





- Don't show dropzone anymore now
- Don't show preview of images but just rounded rectangles for visual inbound

Tag API

- Post /api/images
  → Add multipart field for tags
  ✓ no point having separate call to set tags straight after upload
- Add tags to /api/images (GET)
- PATCH /api/images/{id}/tags
  → take new tags, delete all existing ones
  → its the clients responsibility to send along existing tags if only adding some

Tag UI

- Shortcut to enter selection UI
- Select any image currently displayed
- Change search term to also view others
- Indicator for selection
- Hold shift + click to select all between last and current selected (in current view)
- (- Consider UI for single ~~tag~~ image edit)

→ Adding tags to the upload endpoint duplicates logic

- Also, bulk endpoint will be needed anyway

→ Also, the work to check if tags exist etc. are repeated for every call → Parallel upload, definitely would need transactions here

⇒ We only do style bulk call at the end of upload

Tag computation

image	tags
1	7, 9, 10
2	7, 8, 9, 10
3	5, 6, 7, 8, 10
4	7, 9

tag input if these images are selected

~~7~~ 8 ~~9~~ 10 5 6

we change input to this:

8 10 5 6 11

→ tags we want after mutation

image	tags	change
1	10, 11	-7, -9, +11
2	8, 10, 11	-7, -9, +11
3	5, 6, 8, 10, 11	-7, +11
4	11	-7, -9, +11

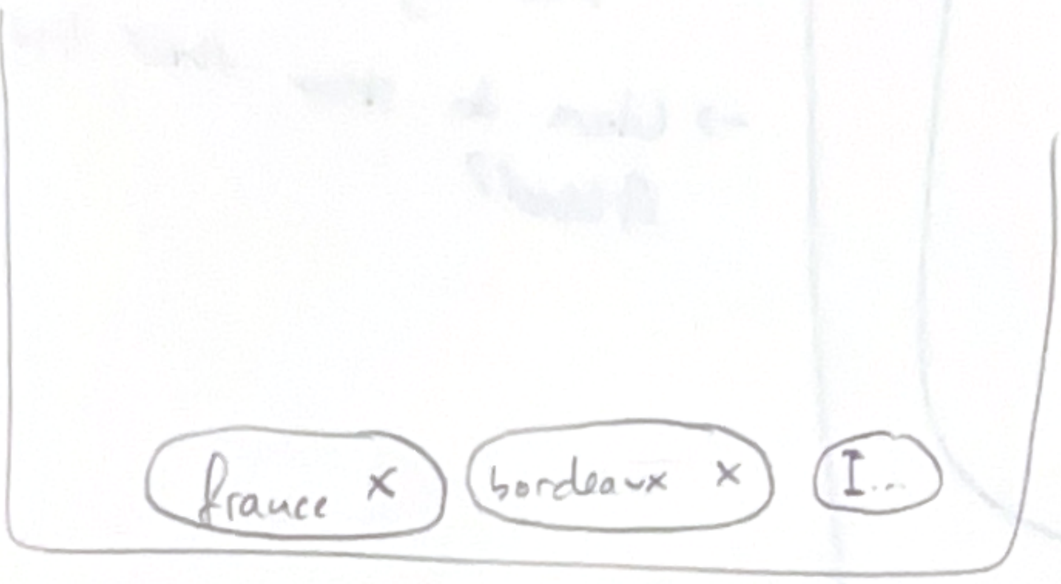
Steps

1. figure out original tag set 1Q
2. compute additions and removals to wanted tag set
3. remove removals 1Q
4. add additions 1Q

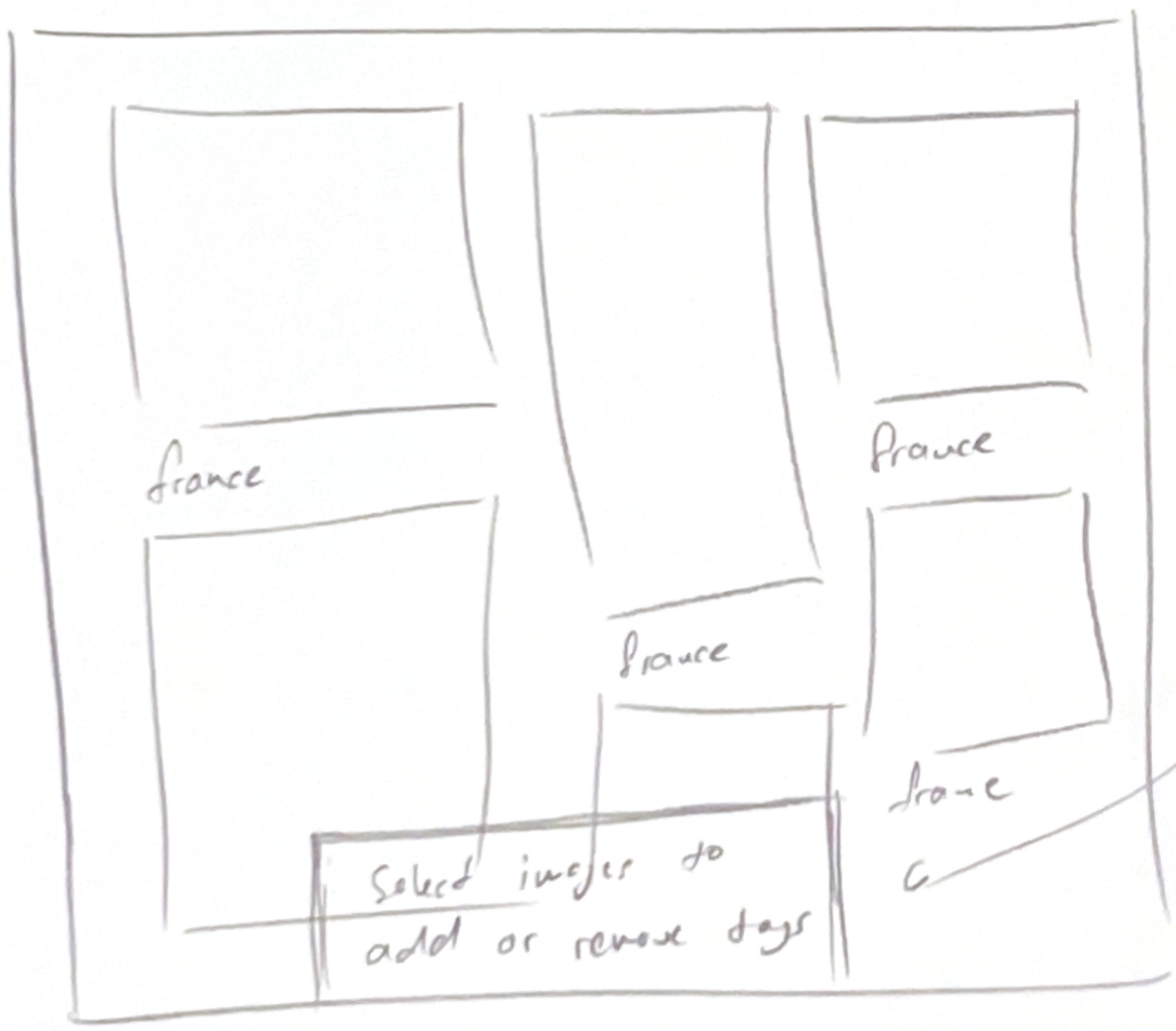
→ 3 Queries

→ Actually split into two endpoints:

- create tags (add)
- remove tags



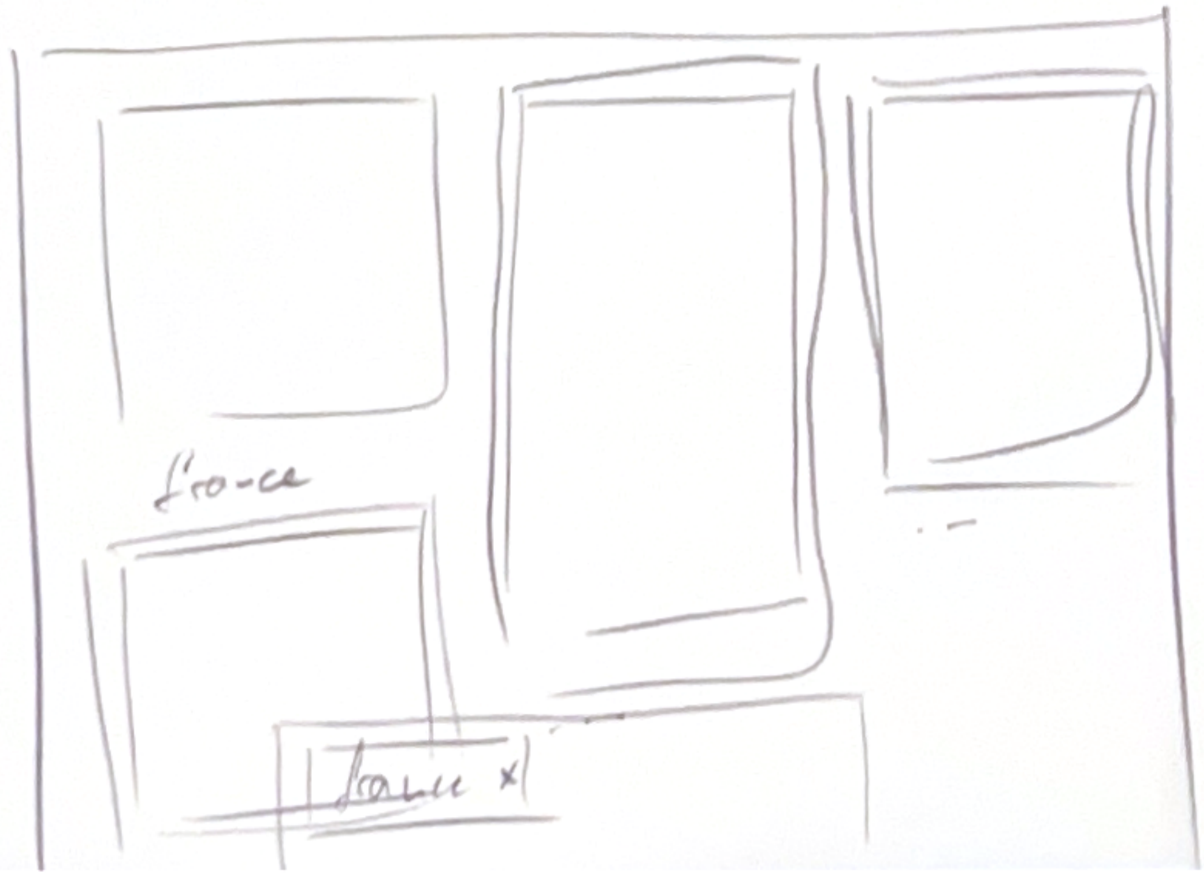
Tag UI v?



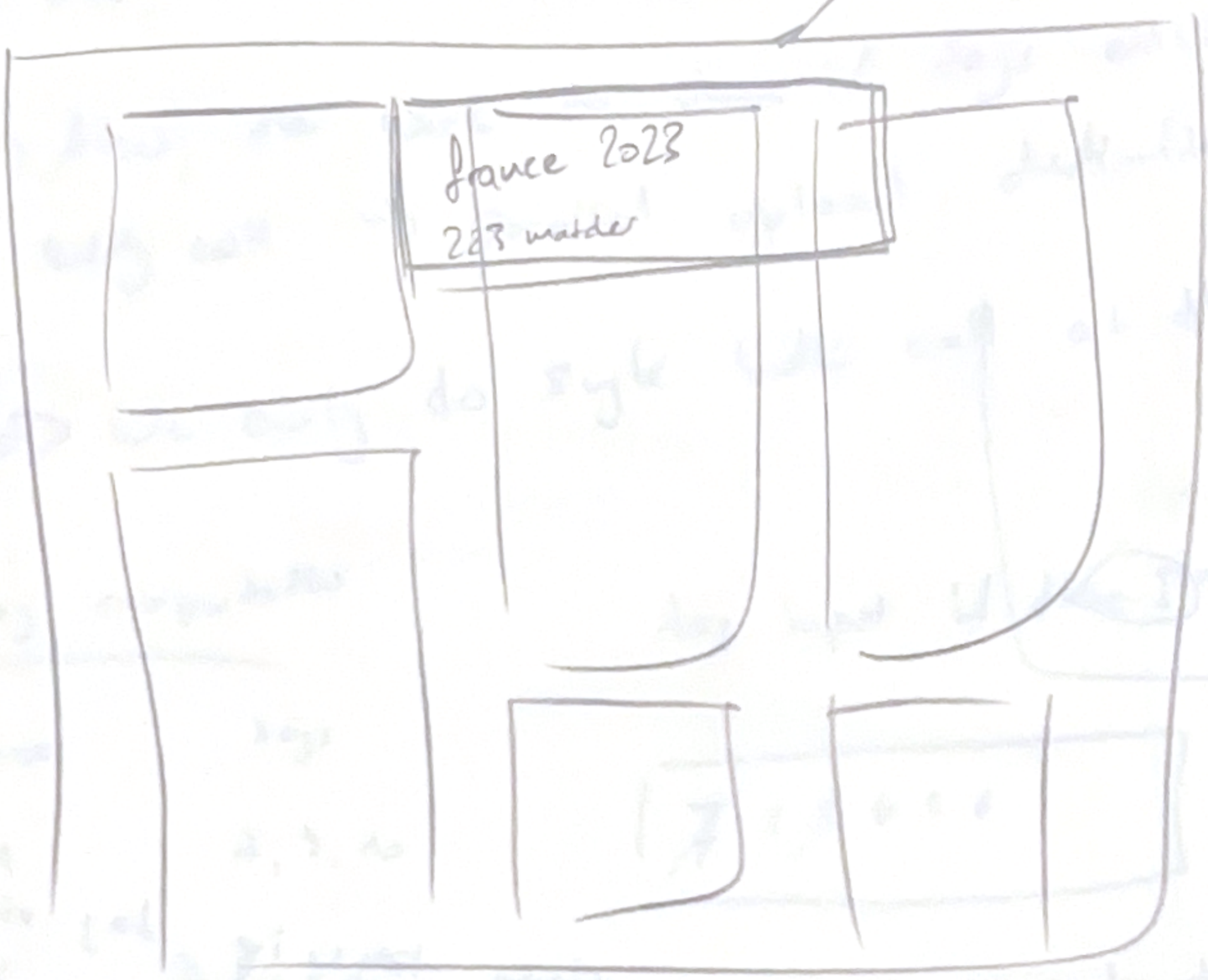
show tags in tag view

Click on image in this view selects, doesn't open lightbox

floating UI element?
→ don't give it to the bottom of the page



Search UI



→ hit ~~Click~~ auto to search
→ already done in background

→ where to show that list is filtered?