6.104 · Software Design · MIT · Fall 2025

modularity case studies

Daniel Jackson

your goals for today's class

deepen your appreciation of modularity
through examples from Zoom and Spotify

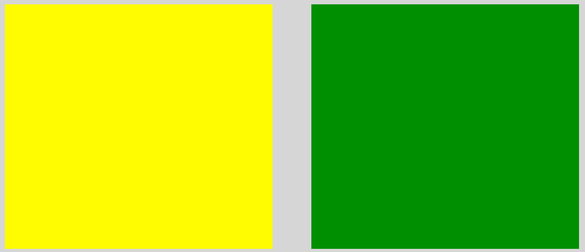
understand impact of concept modularity on code
how concept functions can "cross cut" traditional code

recognize synergy in concept design
when two concepts bring more than the sum of their values

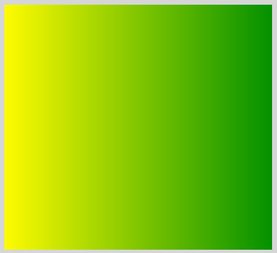
modularity
reviewing criteria

defining modularity

separation
a single module doesn't conflate unrelated functions

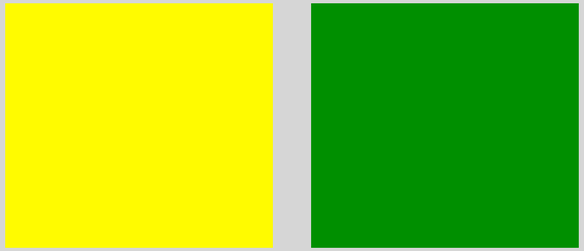


separated: not conflated

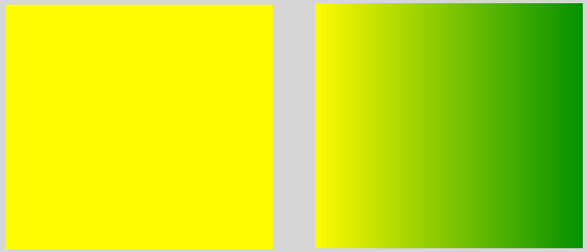


conflated

completeness
a single module contains all of a function's behavior

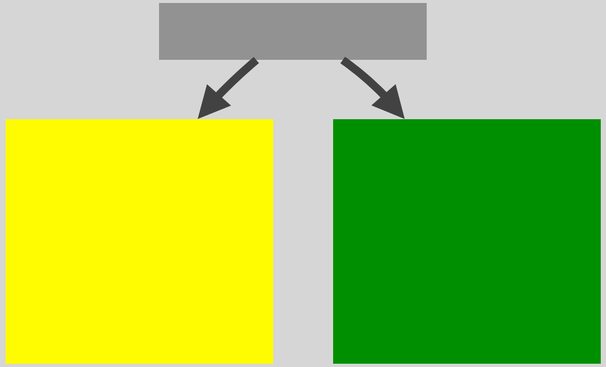


complete: not fragmented

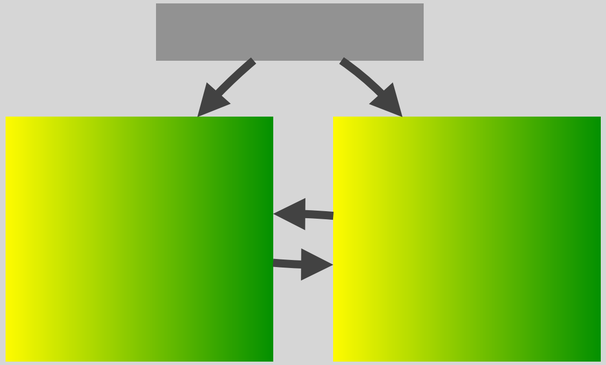


fragmented

independence
one module doesn't rely on another



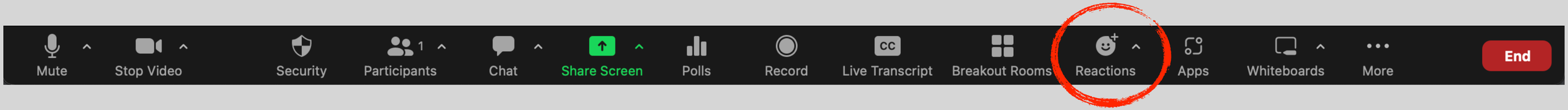
independent

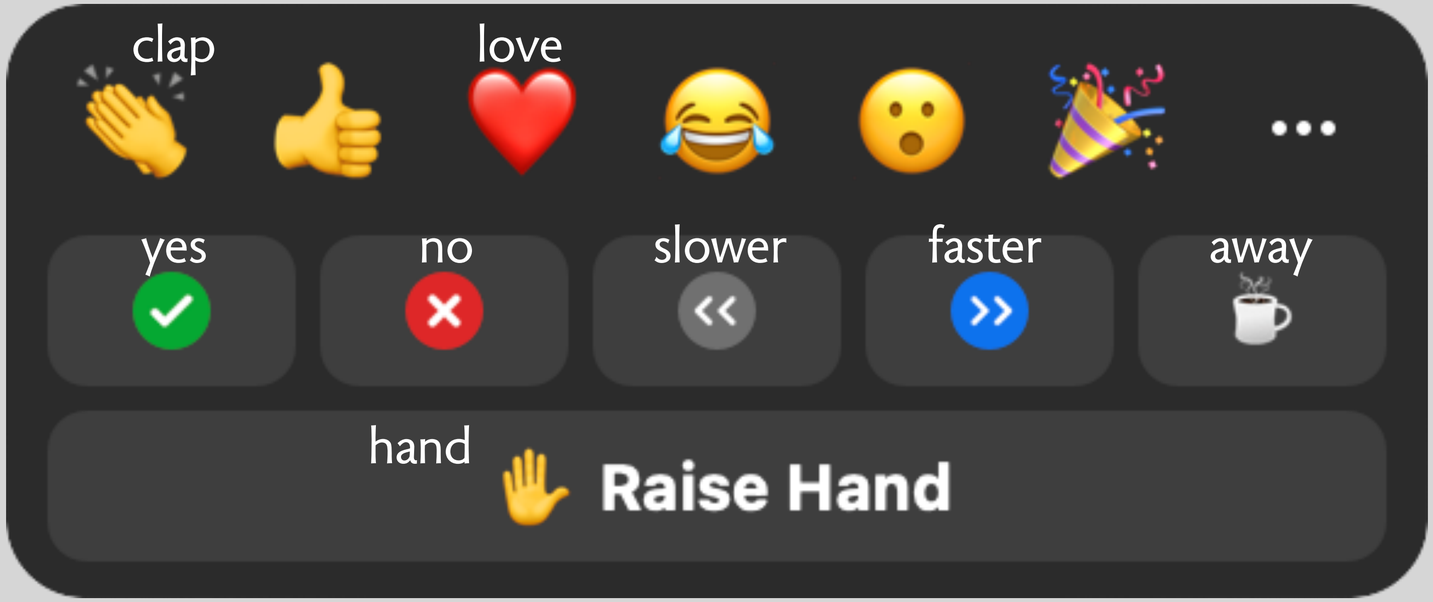


dependent

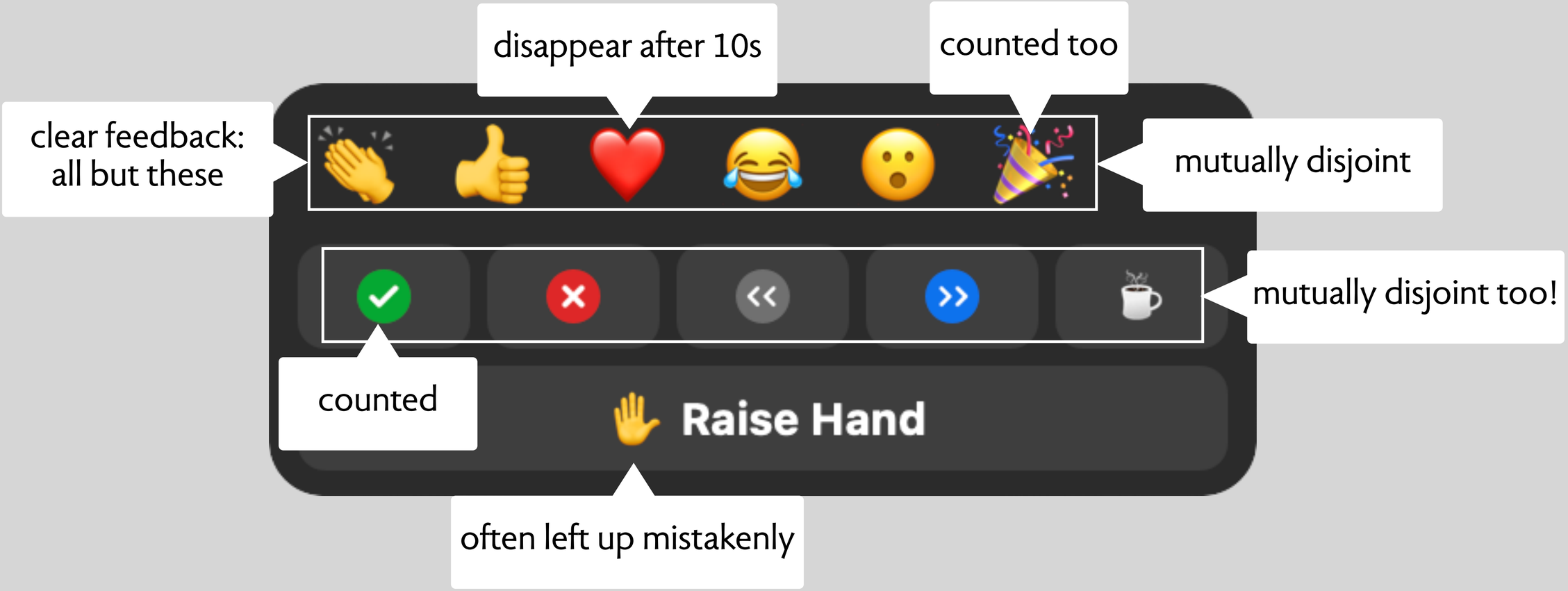
a case study
reactions in Zoom

Zoom's reactions





anomalous behaviors



functions by reaction type

Reaction	Disappears	Counted	Cancel by host
Emojis	✔	(✔)	
Yes/no		✔	✔
Slow/speed		✔	✔
Away		(✔)	(✔)
Hand		(✔)	✔

✔ yes

(✔) yes, but should probably be no

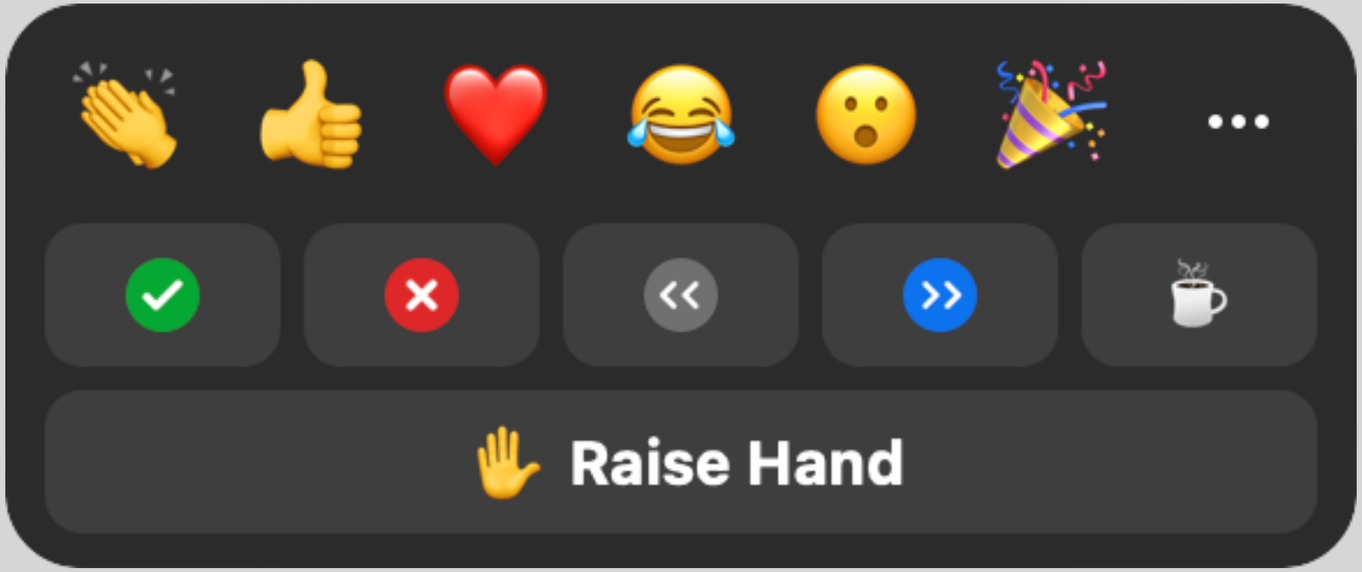
disjointness of reaction types: my take

Reaction	Emojis	Yes/no	Slow/speed	Away	Hand
Emojis	✔				
Yes/no		✔	(✔)	(✔)	(✔)
Slow/speed		(✔)	✔	(✔)	(✔)
Away		(✔)	(✔)	✔	(✔)
Hand		(✔)	(✔)	(✔)	✔

✔ yes

(✔) yes, but should probably be no

an exercise: can you do better?



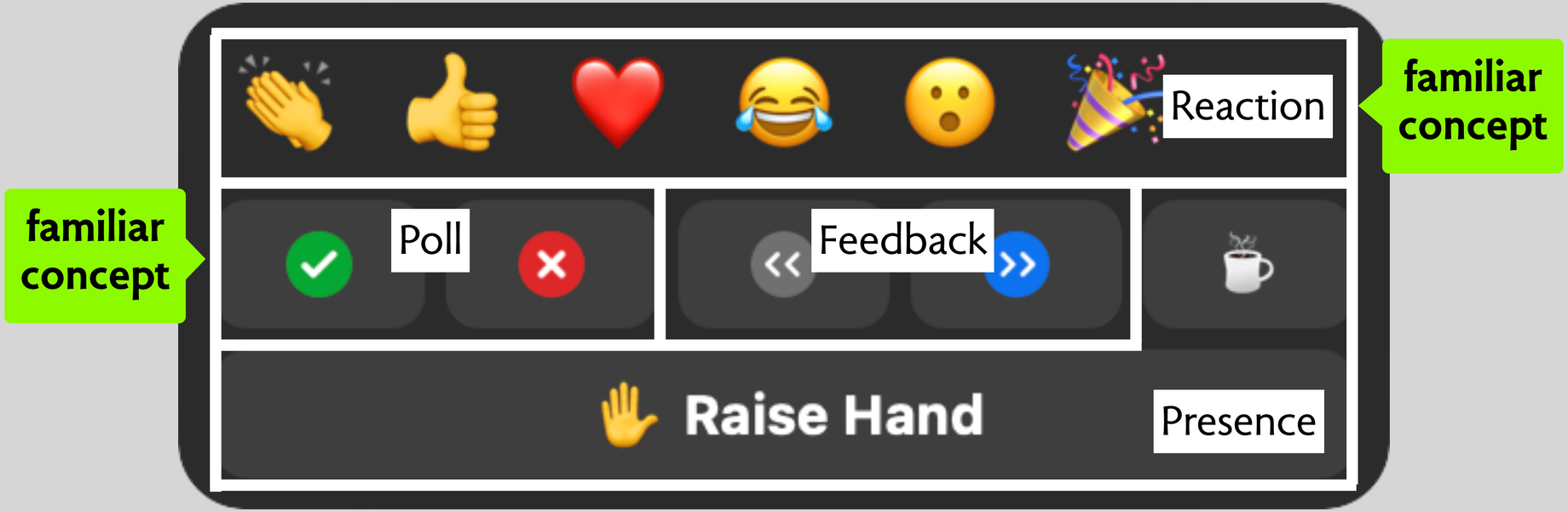
goals

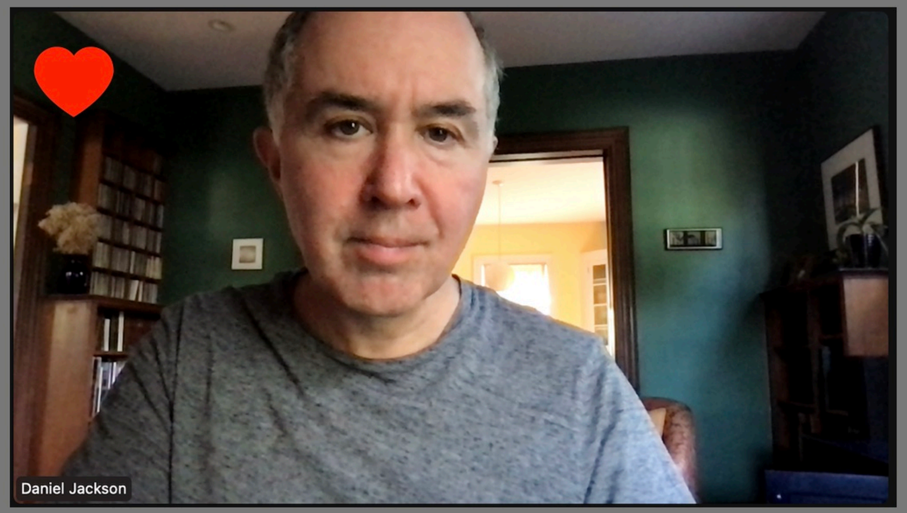
break the behavior into a small set of concepts (in outline)

use familiar concepts whenever possible

apply modularity criteria: separation & completeness

my take: splitting into coherent concepts





Presence

Request to speak

Speaking

Watching/listening

I'm away

Chat

Audience

Everyone ▾

Type here…

Reaction

Recent emoji

Other emoji

…

Feedback

Slow down

Just right

Speed up

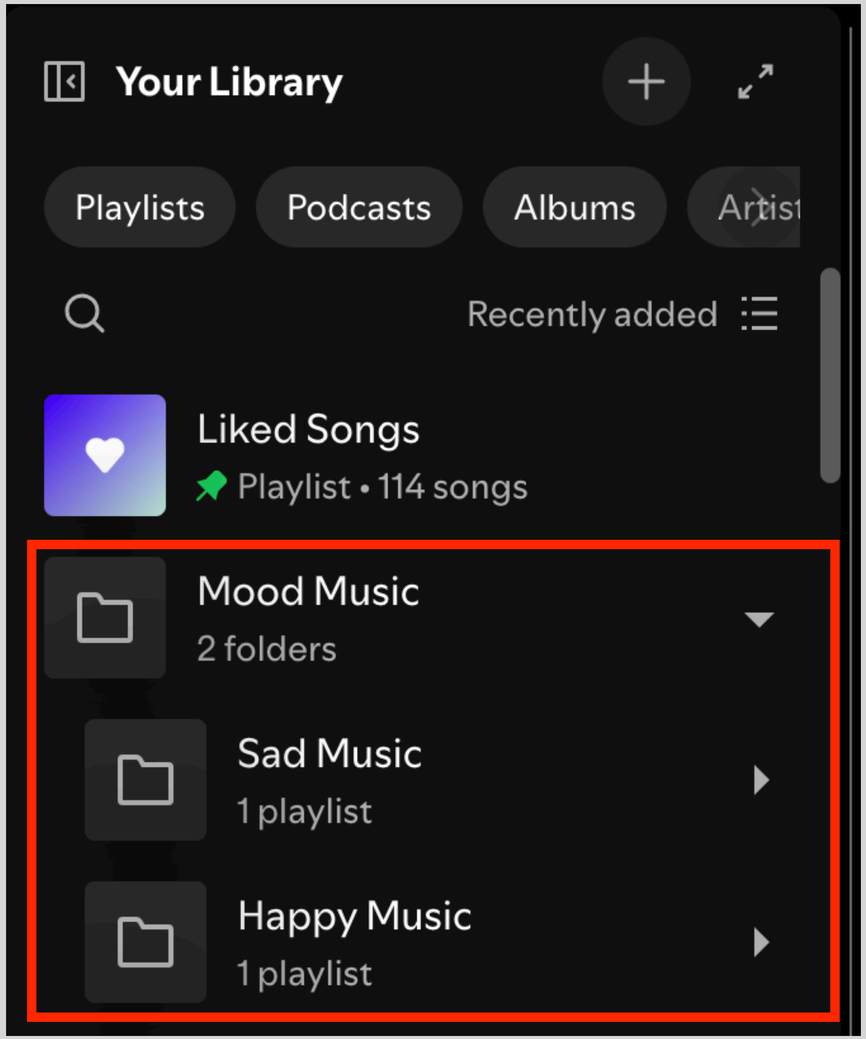
16

24

9

case studies
in spotify

playing with spotify folders



what happens when you add a playlist to a folder?

it inserts the playlist into the folder

when you add an album (or a song) to a folder?

it creates a new playlist contain the song(s)

it inserts the playlist into the folder

does it publish the playlist making it public?

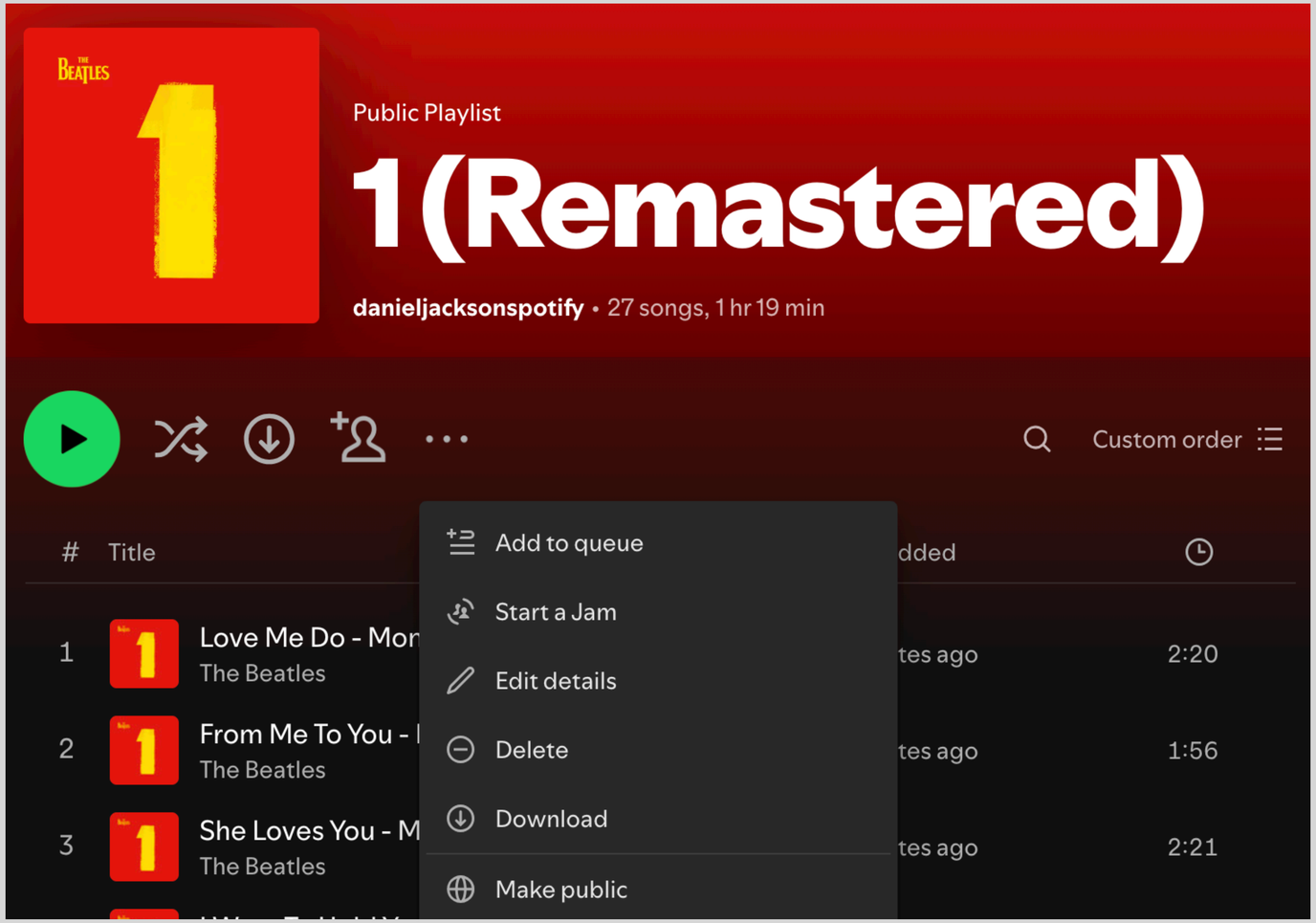
when you add a podcast to a folder?

when you add an artist to a folder?

when you add liked songs playlist to a folder?

you can't

huh?


Public Playlist
1 (Remastered)
danieljacksonspotify • 27 songs, 1 hr 19 min
Custom order
Title
Add to queue
Start a Jam
Edit details
Delete
Download
Make public
1 Love Me Do - Mor
The Beatles
2:20
2 From Me To You -
The Beatles
1:56
3 She Loves You - M
The Beatles
2:21

your turn

how do these inconsistencies impact users?

do they really matter? do they introduce friction?

do they prevent users from doing things they'd like to do?

what are the concepts here?

what actually is the folder concept about? its purpose? state?

how might you improve this design?

what concepts would you have?

can you achieve simplicity & flexibility at once?

a concept analysis

concept Folder [Item]

elements are generic: any kind

purpose

organize items in a hierarchy

principle

after you create a folder and insert elements into it, you can move the folder into another folder and all the elements will still belong to it

state

a set of Folders with
- a name String
- a contained set of Folders
- an elements set of Items

actions

insert (i: Item, f: Folder)

insert just adds item to elements

...

what we're expecting

concept PlaylistTree

purpose

organize playlists in a hierarchy

principle

after you create a folder and insert playlists into it, you can play the whole folder, which plays the playlists in order

state

a set of Folders with
- a name String
- a contained set of Folders
- an elements set of Playlists

element can't be a song either!

actions

insert (p: Playlist, f: Folder)

insert doesn't let you choose order

...

the actual concept

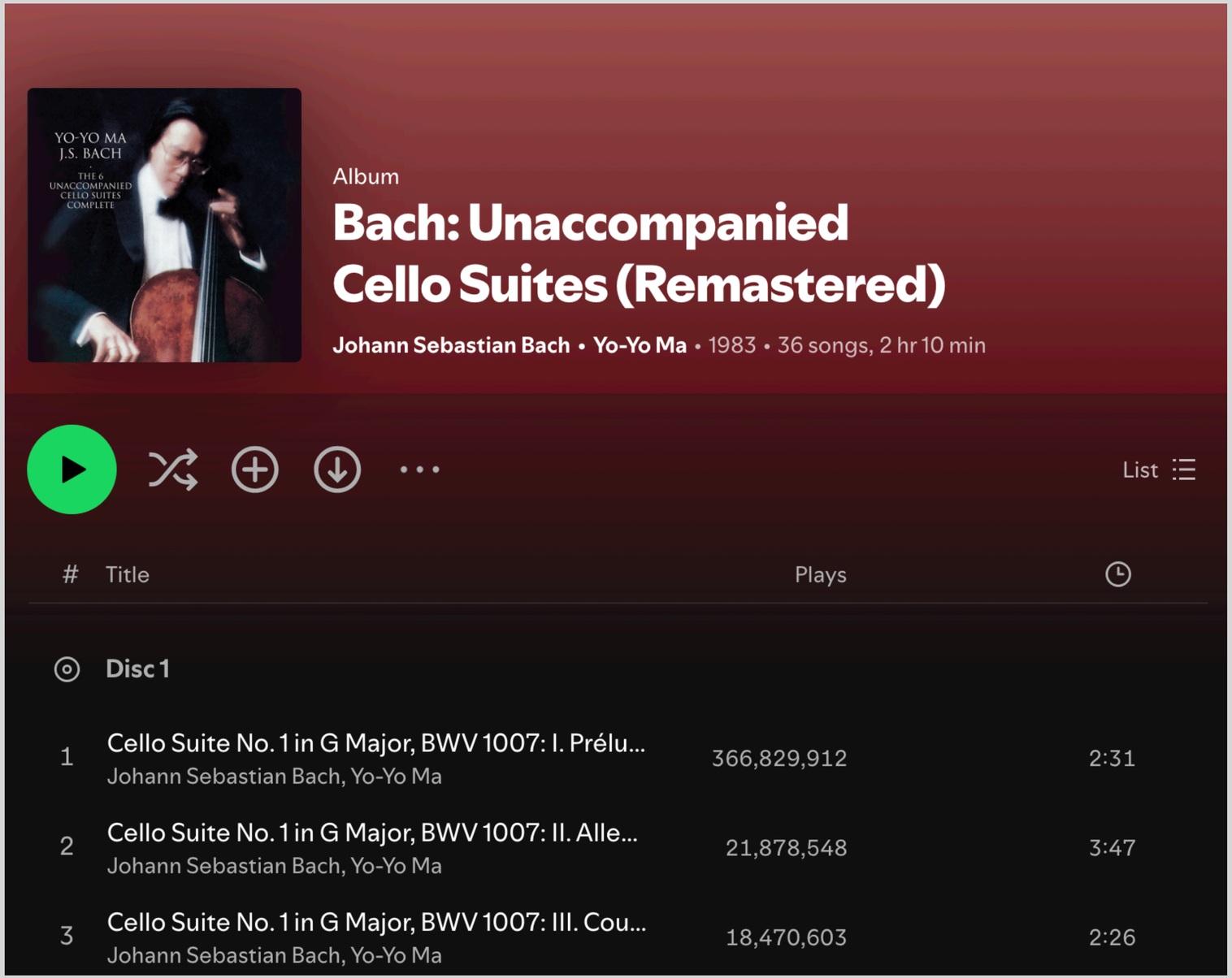
an awkward hybrid concept

standard folders
can put anything in a folder
no conversion to playlists
no “playing” of folders

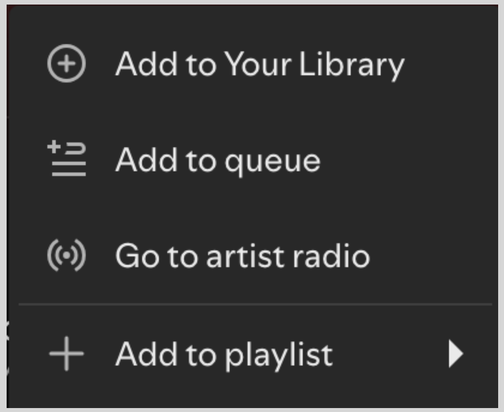
spotify folder
can only add playlists
can only share playlists
can play but can’t set order

nested playlists
can add song at any level
can share at any level
can set order of songs & lists

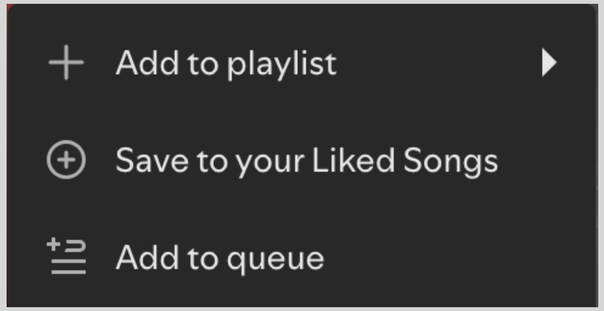
what is a playlist?



clicking on three dots for album



clicking on three dots for song



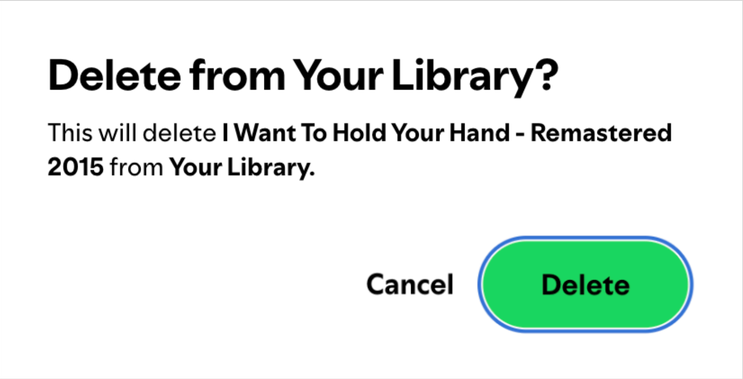
what happens when you delete a song?

if a song is in any of your playlists, then it's in your library
so deleting a playlist can remove songs from your library

if a song is in a playlist and an album in your library
then deleting it from the playlist will not remove it from the library

if a song is in two playlists
then deleting it from one will not remove it from the library

"saving" a song to liked songs
adds it to a special playlist called "liked songs"



what spotify says when you delete a song
even when it's in more than one playlist

your turn

how do these inconsistencies impact users?

do they really matter? do they introduce friction?

do they prevent users from doing things they'd like to do?

what are the concepts here?

what actually is the playlist concept for? its purpose?

how might you improve this design?

can you achieve simplicity & flexibility at once?

one way to redesign

concept Library [User, Song]

purpose

save songs & albums for easy access

state

a set of Users with
  a set of Songs
  a set of Albums
a set of Albums with
  a set of Songs

actions

save (u: User, s: Song)
save (u: User, a: Album)
discard (u: User, s: Song)
discard (u: User, a: Album)
...

songs & albums may overlap

how are Library.discard and Playlist.remove synchronized?

concept Playlist [User, Song]

purpose

organize songs into listening lists

state

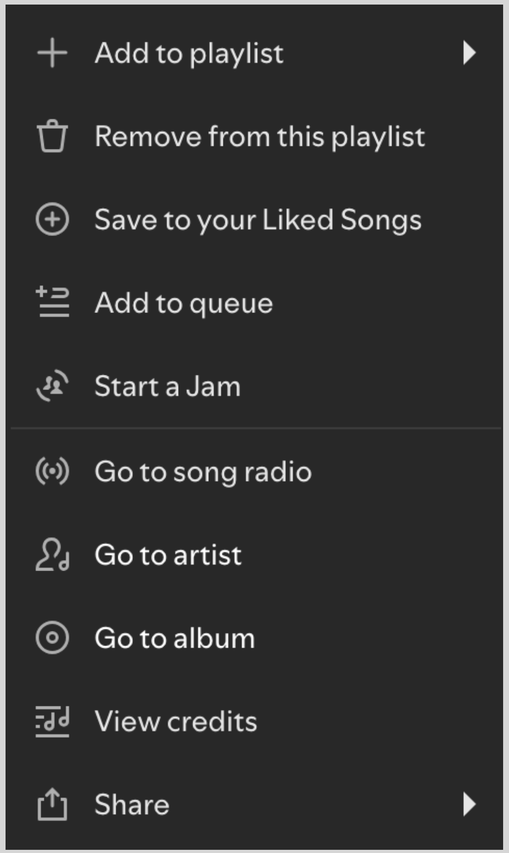
a set of Users with
  a set of Playlists
a set of Playlists with
  a seq of Songs

actions

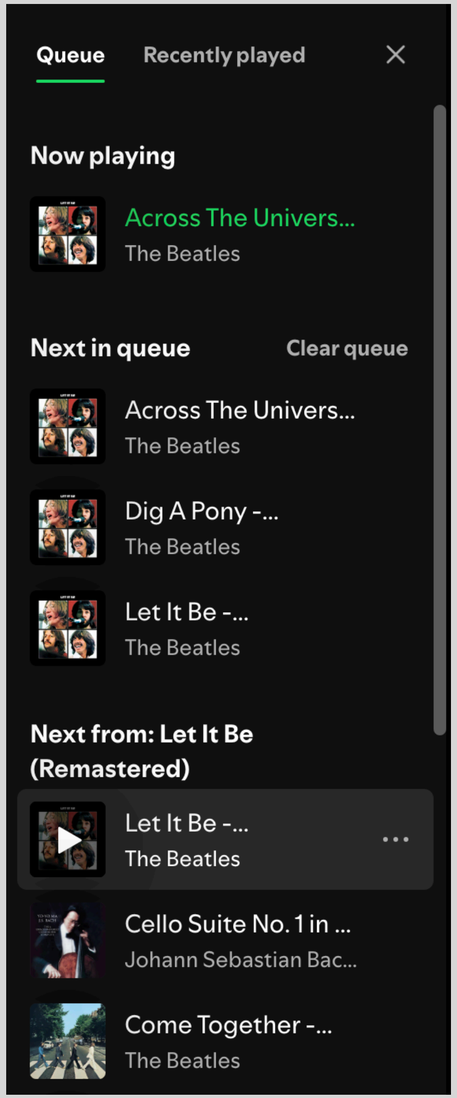
add (p: Playlist, s: Song)
remove (p: Playlist, s: Song)

what will sync need to check before Playlist.add happens?

playing with the spotify queue



when you click on a song



looking at the queue

when you start a song:
replaces **now playing**

when you add to queue:
adds to end of **next in queue**

when you start a song:
replaces **next from** (using context)

you can also **move songs**
between sections

your turn

what about a conventional queue concept?

standard queue is FIFO: first in, first out

when do the songs you add to the queue play?

what would happen when you switched on autoplay?

what's really going in spotify's queue

concept Queue [Song]
purpose
  let users manually select song order
state
  a set of Users with
    a seq of Songs
actions
  enqueue (u: User, s: Song)
  clear (u: User)
  ...

concept Feed [Song]
purpose
  provide endless stream of songs
state
  a set of Users with
    a seq of Songs
actions
  populate (u: User, ...)
  dequeue (u: User, ...)

sync drawSongFromQueue
when PlayingSong.ends (u)
where
  Queue: first song for u is s
then
  PlayingSong.setSong (u, s)
  Queue.dequeue (u)

sync drawSongFromFeed
when PlayingSong.ends (u)
where
  Queue: no songs in queue for u
  Feed: first song in feed for user is s
then
  PlayingSong.set (u, s)
  Feed.dequeue (u)

concept PlayingSong
purpose
  play songs
state
  a set of Users with
    an optional playing Song
actions
  set (u: User, s: Song)
  start (u: User)
  system ends (u: User)

lessons

genericity

familiarity

lack of modularity

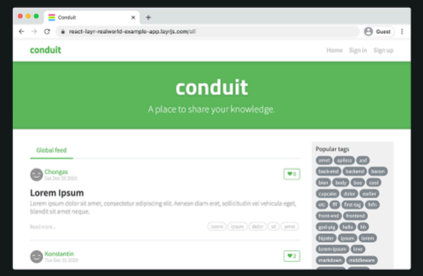
unpredictable behavior
when will deleting a song from a playlist remove it from your library?

how much do these issues affect novice users? experts?

why hasn't Spotify fixed some of these problems?

a RealWorld case study

how concept modularity impacts code



The mother of all demo apps

See how the exact same application is built using different libraries and frameworks.

Frontend | Backend | Fullstack

LANGUAGES

- All
- TypeScript
- JavaScript
- Kotlin
- ClojureScript
- Elm
- PureScript
- Rust
- C#
- Dart
- Mint
- Swift
- ReScript

Android Native MOBILE coding-blocks-archives/Conduit_Android_Kotlin	Kotlin
Android Native + Retrofit + Jetpack MOBILE Marvel999/Conduit-Android-kotlin	Kotlin
Angular khaledosman/angular-realworld-example-app	TypeScript
Angular AndyT2503/angular-conduit-signals	TypeScript
Angular iancharlesdouglas/ng-realworld-ssr	TypeScript
Angular + NgRx + Nx	TypeScript

RealWorld example app

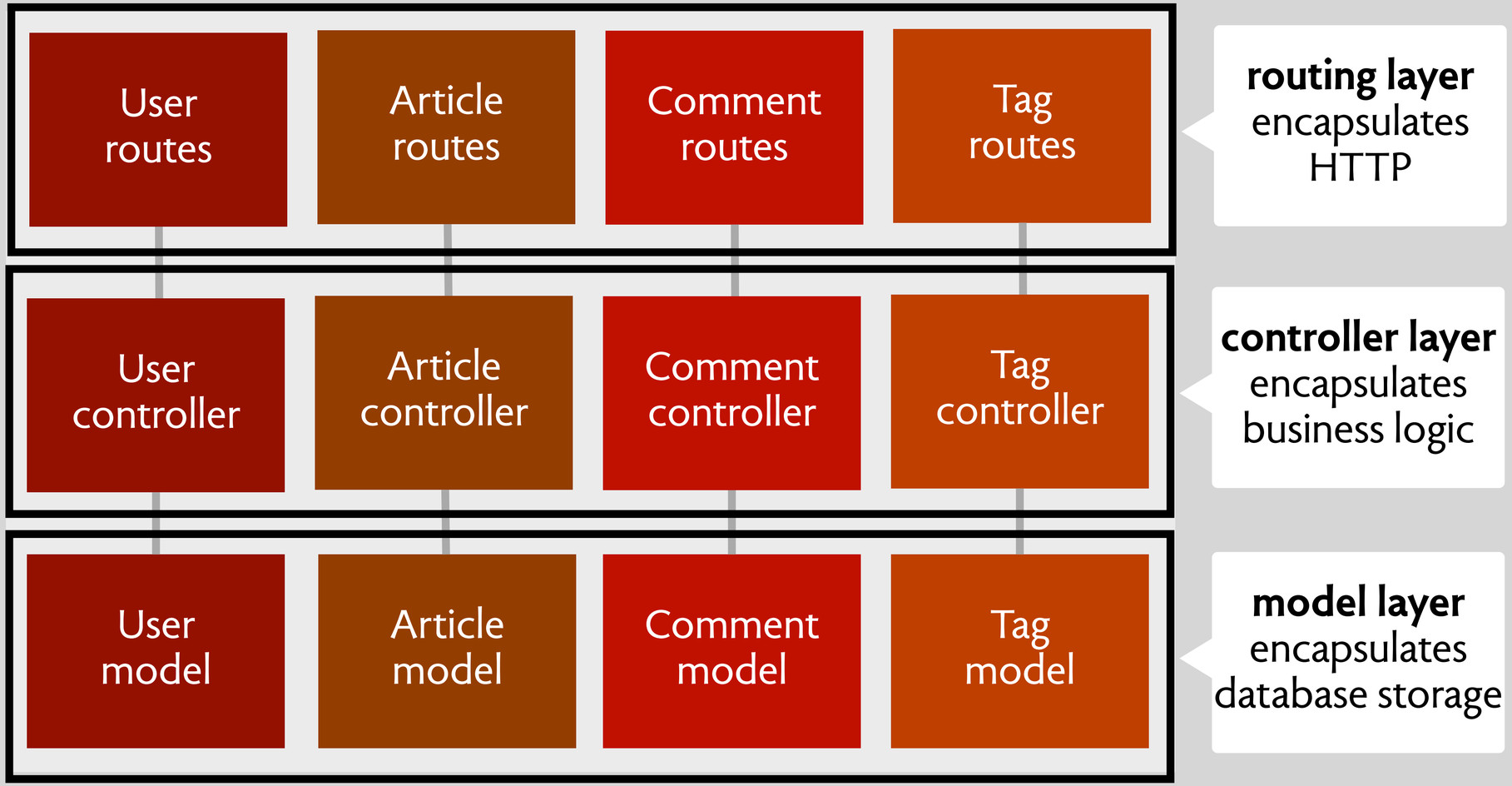
> Express.js + MongoDB + JavaScript codebase containing real world examples (CRUD, auth, advanced patterns, etc) that adheres to the RealWorld spec and API.

Demo **RealWorld**

This codebase was created to demonstrate a fully fledged fullstack application built with **Express.js + MongoDB + JavaScript** including CRUD operations, authentication, routing, pagination, and more.

We've gone to great lengths to adhere to the **Express.js + MongoDB + JavaScript** community styleguides & best practices.

https://github.com/winterrrrrff/realWorld-server

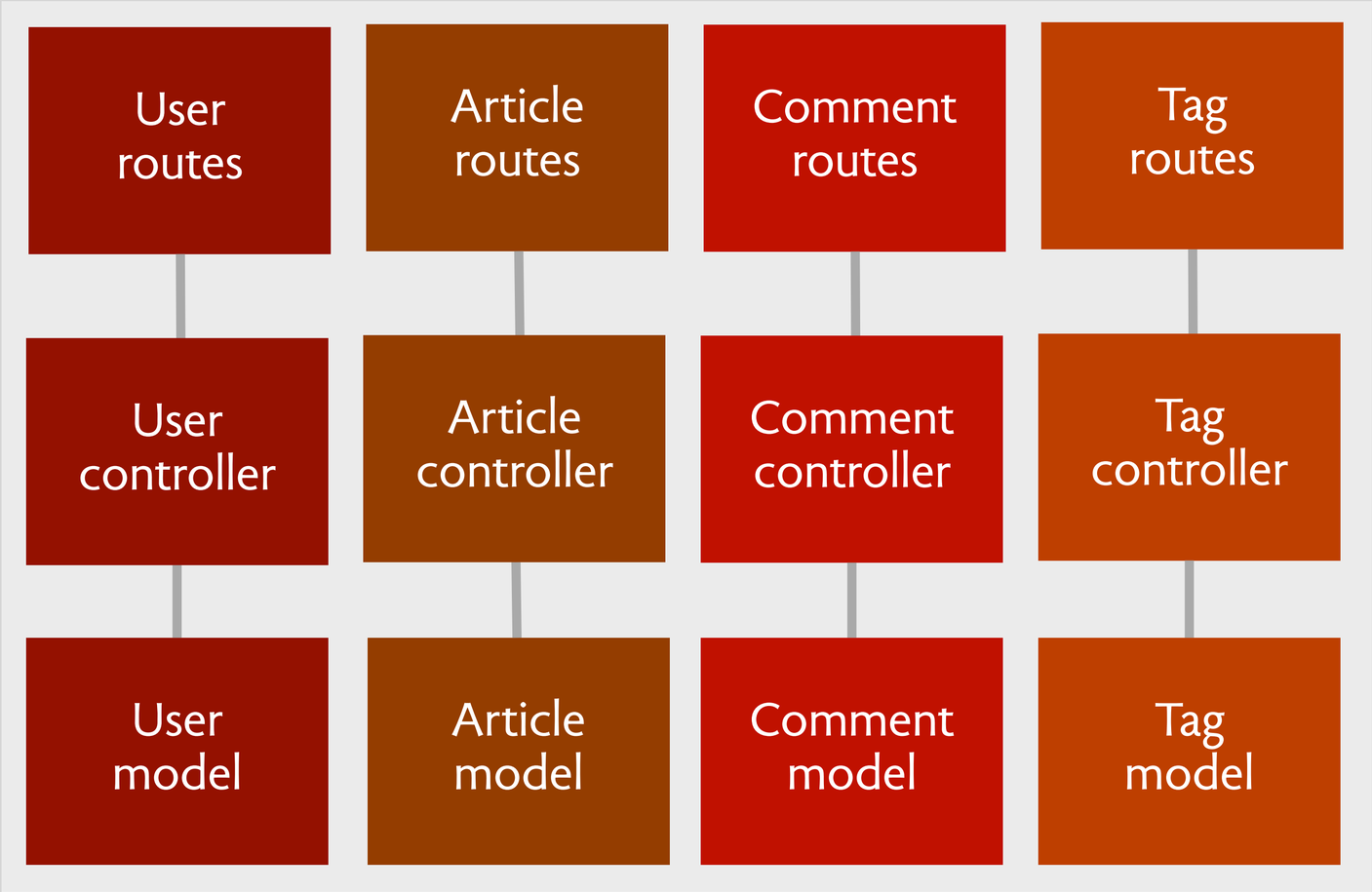

User routes
Article routes
Comment routes
Tag routes
routing layer
encapsulates HTTP
User controller
Article controller
Comment controller
Tag controller
controller layer
encapsulates business logic
User model
Article model
Comment model
Tag model
model layer
encapsulates database storage

where does it go?
functions seem to have natural homes

allow articles without titles

allow longer comments

let user change name

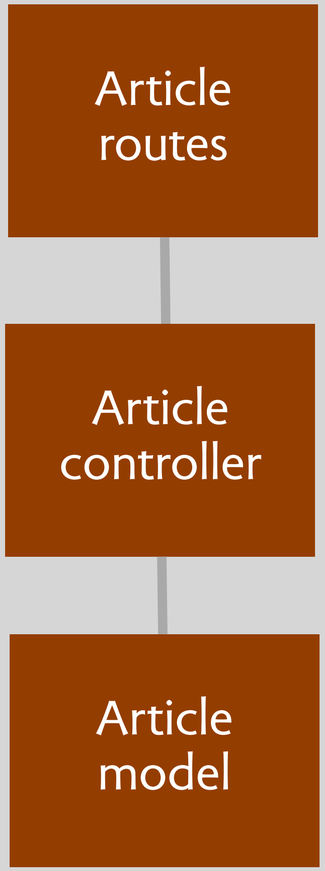


an example: article-specific function

```
router.post('/', verifyJWT, articleController.createArticle);
```

```
createArticle = asyncHandler((req, res) => {
 { title, description, body } = req.body.article;
 if (!title || !description || !body)
    res.status(400).json({message: "All fields are required"});
 article = Article.create({ title, description, body });
 article.save()  ...});
```

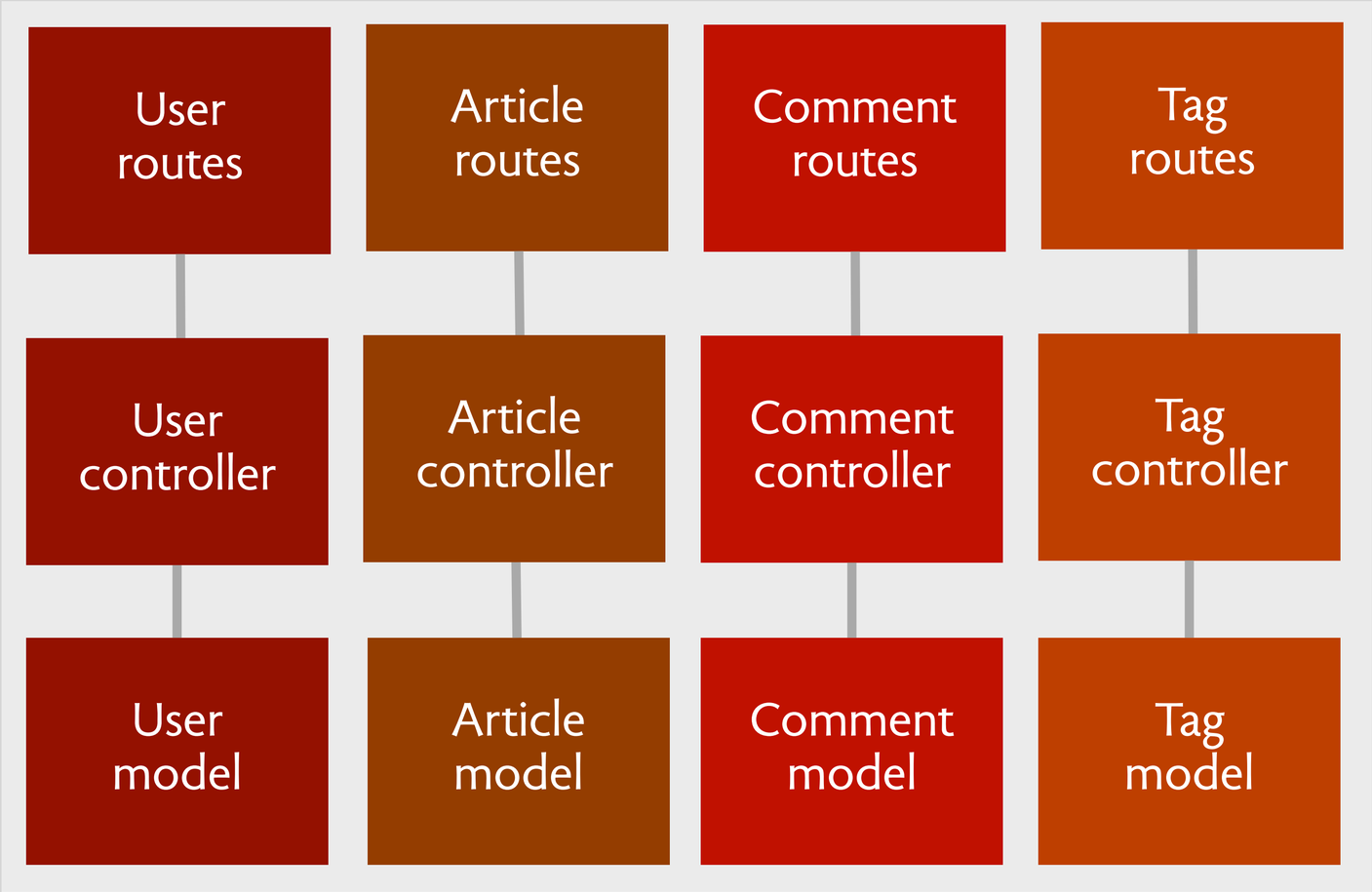
```
Article = new mongoose.Schema({
   title: {type: String, required: true},
   description: {type: String, required: true},
   body: {type: String, required: true}...})
```



what's not great about this code?

what about favoriting?

where does it go?
favorites associate
users with articles

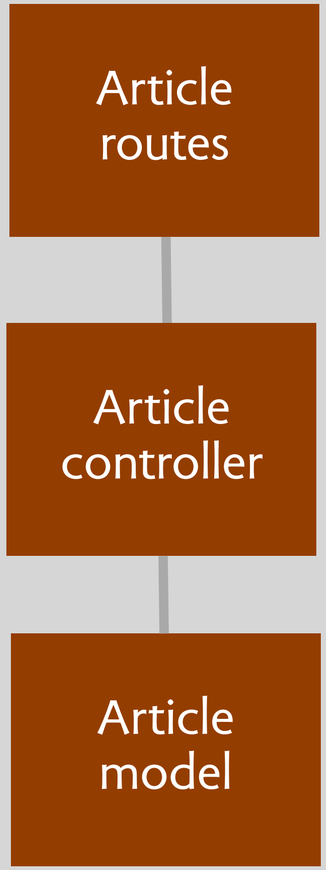


an example: a cross-object function (1)

```
router.post('/:slug/favorite', verifyJWT, articleController.favoriteArticle);
router.delete('/:slug/favorite', verifyJWT, articleController.unfavoriteArticle);
```

```
favoriteArticle = asyncHandler((req, res) => {
   loginUser = User.findById(id).exec();
   article = Article.findOne({slug}).exec();
   loginUser.favorite(article._id);
   updatedArticle = article.updateFavoriteCount();
   ... });
```

```
Article = new mongoose.Schema({
   favouritesCount: {type: Number, default: 0}, ... });

Article.methods.updateFavoriteCount = function () {
   favoriteCount = User.count({favouriteArticles: {$in: [this._id]}});
   this.favouritesCount = favoriteCount;
   return this.save(); }
```

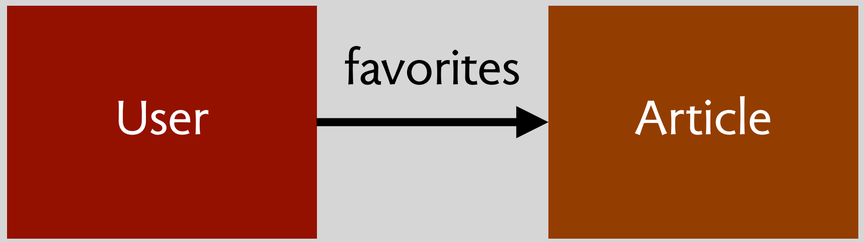


what's not great about this code?

an example: a cross-object function (2)

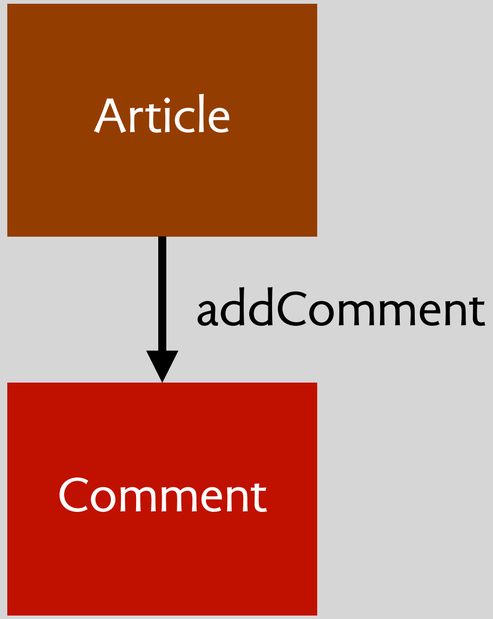
```
User = new mongoose.Schema({
   favouriteArticles: [{
     type: mongoose.Schema.Types.ObjectId,
     ref: 'Article'}],...});

User.methods.favorite = function (id) {
   if(this.favouriteArticles.indexOf(id) === -1)
      this.favouriteArticles.push(id);
   // const article = Article.findById(id).exec();
   // article.favouritesCount += 1;
   // article.save();
   return this.save(); }
```

what's not great about this code?



many features involve >1 object
eg, favorites relates Users to Articles

objects conflate concerns
authentication & following are both in User

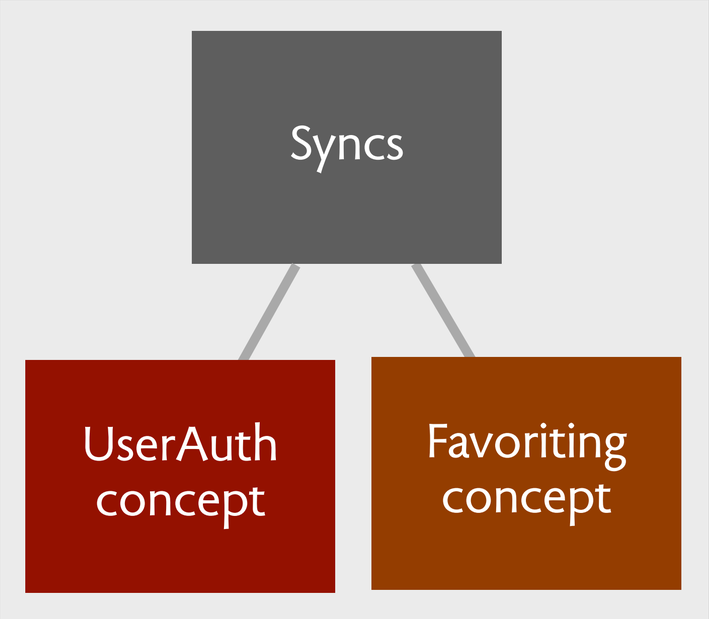


OOP encourages fragmentation
eg, addComment is method of Article

how concept code would work

sync connects UserAuth to Favoriting

```
sync when
  Request.addFavorite (article)
  UserAuth.getCurrentUser (): user
then
   Favoriting.addFavorite (user, article)
```



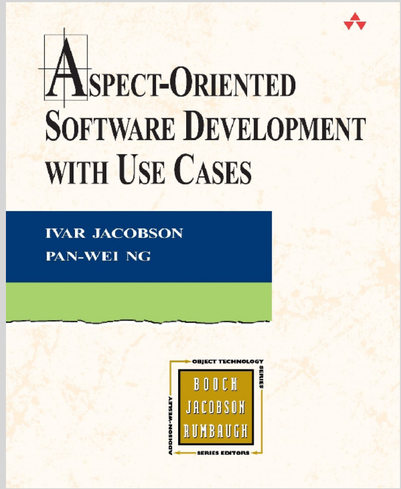

```
getCurrentUser = function () {
   ...
   return user; });
```

no mention of favoriting in UserAuth

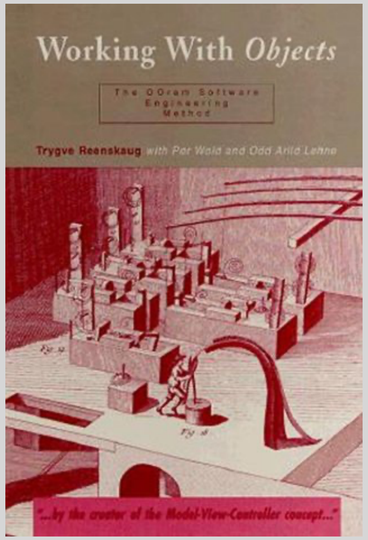
```
addFavorite = function (user, item) {
   favorites.insert (user, item)
  ... });
```

in favoriting code users are just ids

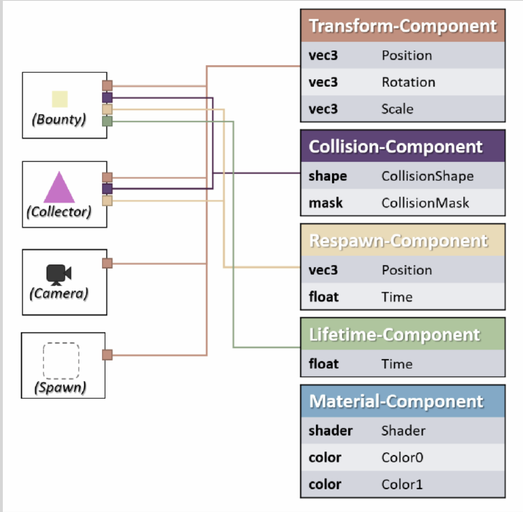
a long history of fixes for OOP's conflation



Aspect-oriented programming
Kiczales et al (1997)



Role-oriented programming
Reenskaug et al (1983)

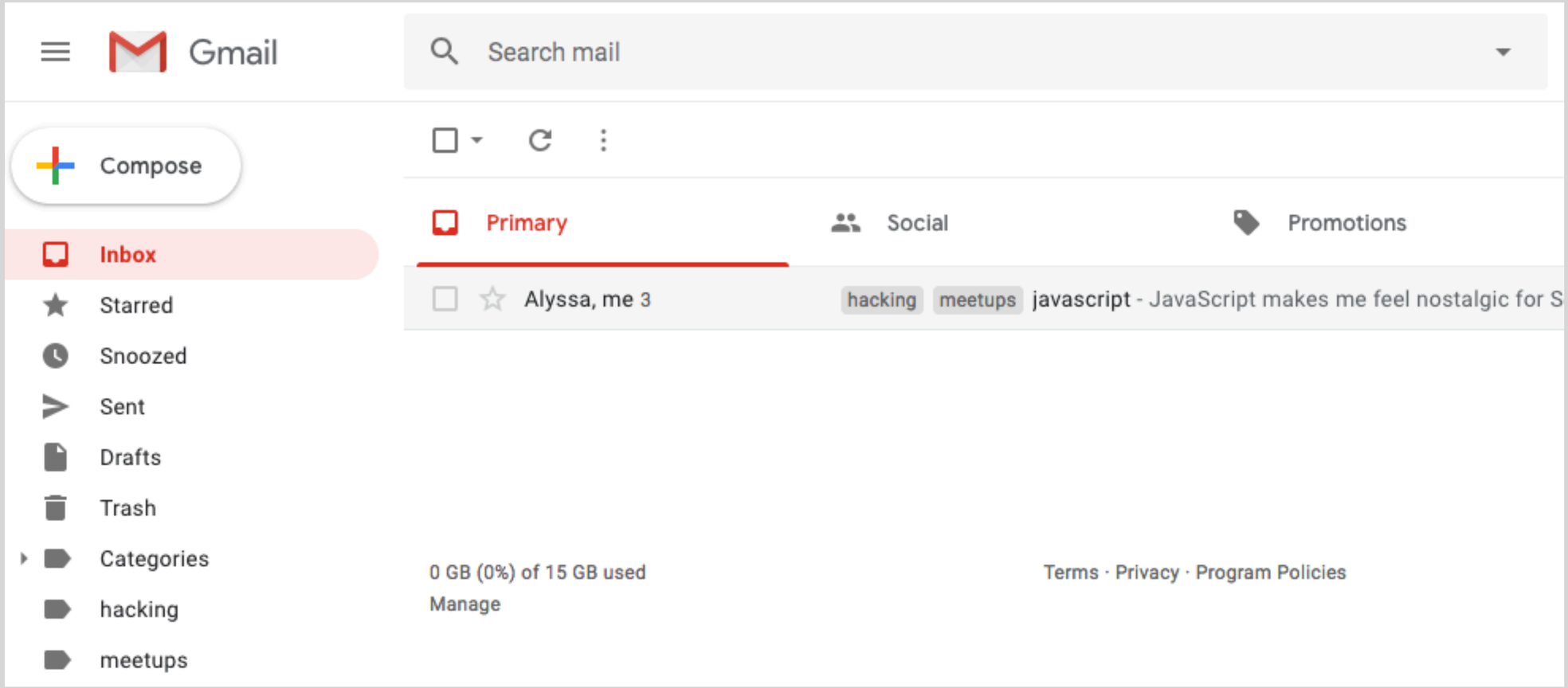


Entity-component system
Scott Bilas et al (2002)

synergies
in concept design

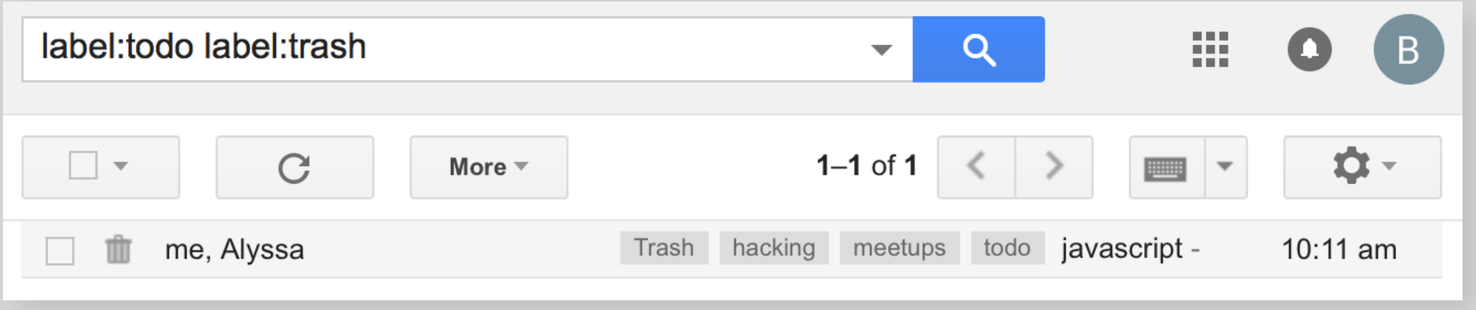
Gmail
labeling

the role of labels in Gmail



just a label lookup

use sent, trash in filters



what other benefits does this design bring?

what's the sync?

concept Trash

purpose allow undeletion

principle if an object is deleted, and the trash is not emptied, it can be restored; if an object is deleted and the trash is emptied, the object is gone but its space is regained

actions
delete (o: Object)
restore (o: Object)
empty ()

concept Labeling

purpose organize items

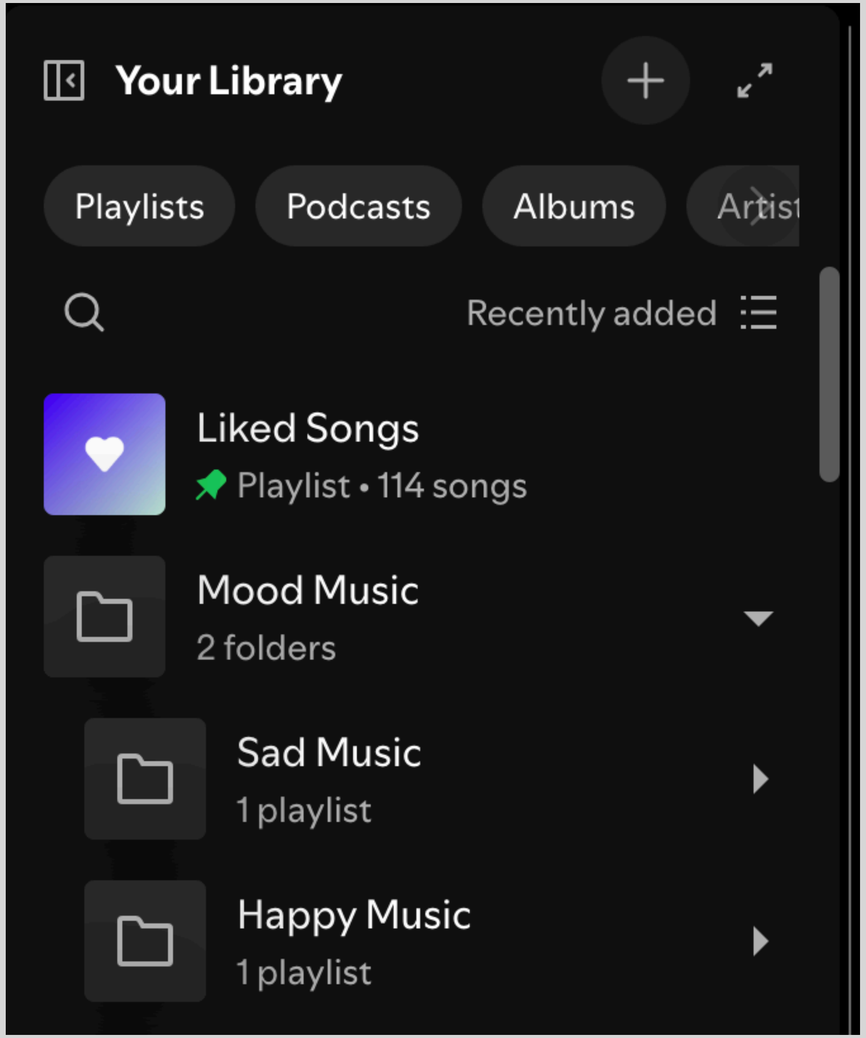
principle if label is added to an item, then filtering on that label will display that item

actions
add (l: Label, i: Item)
remove (l: Label, i: Item)
filter (ls: set Label): set Item

syncs include rules like these and their converse

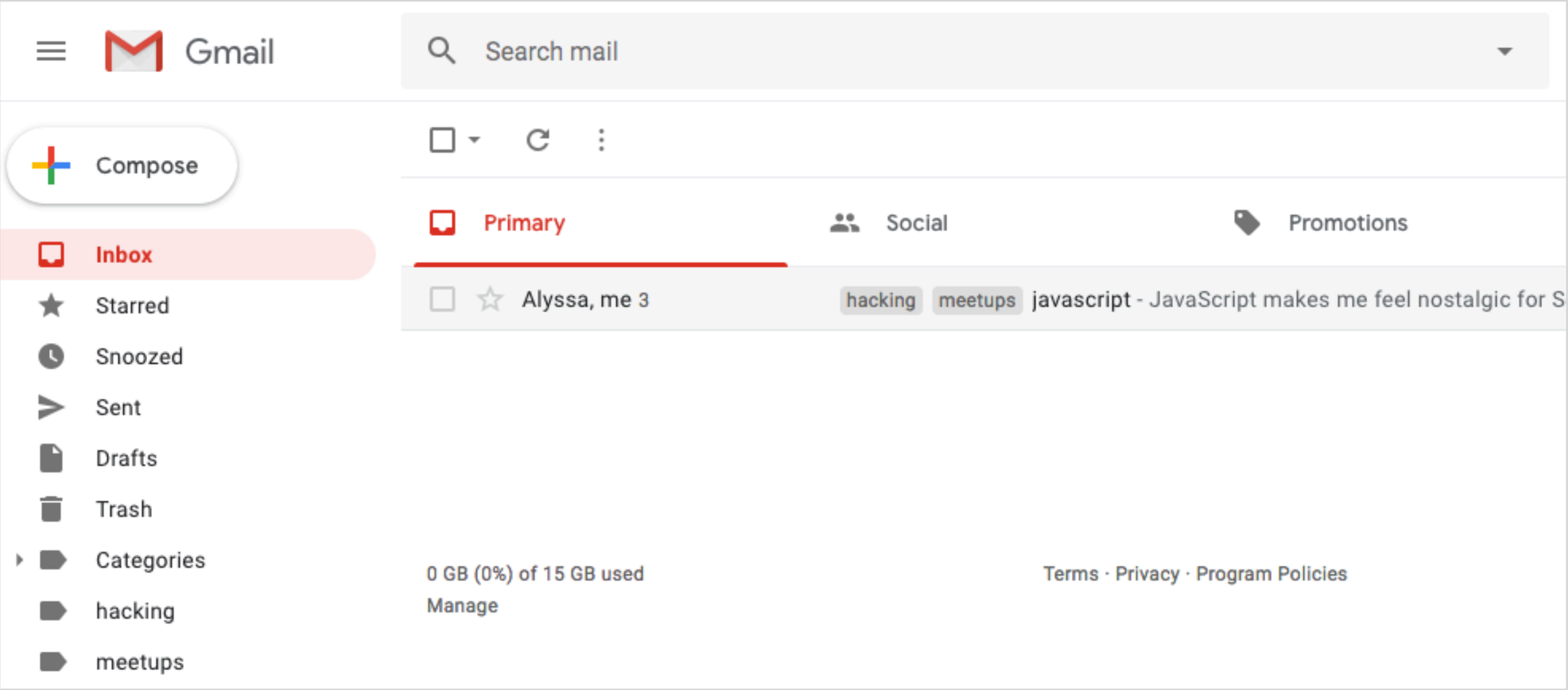
when Trash.delete(i) **then** Labeling.add (‘*trashed*’, i)
when Trash.restore(i) **then** Labeling.remove (‘*trashed*’, i)

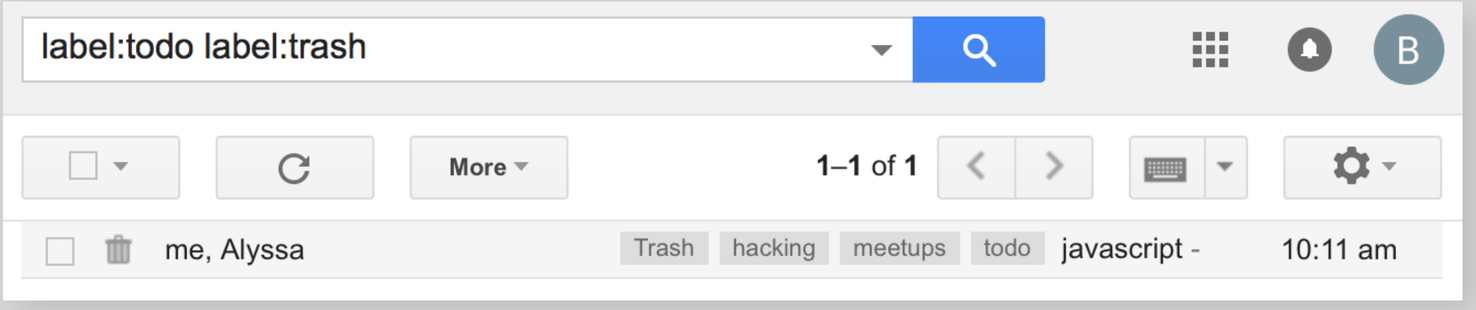
a spotify synergy



Liked songs is a similar synergy
How so?

Gmail complications

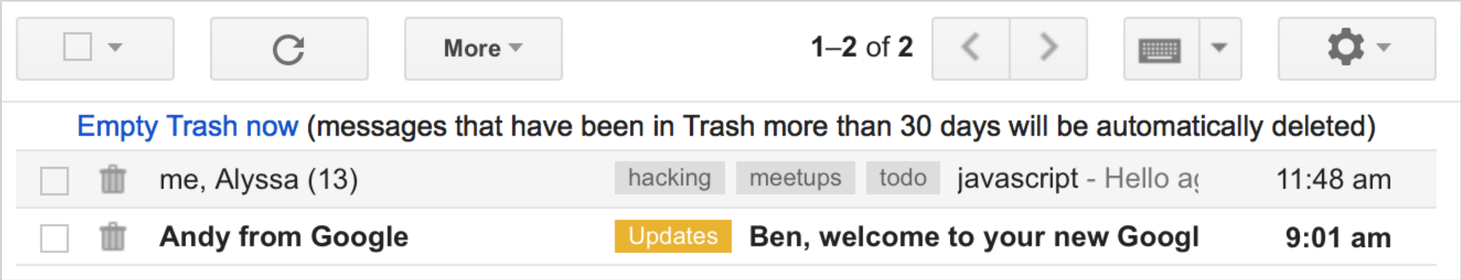




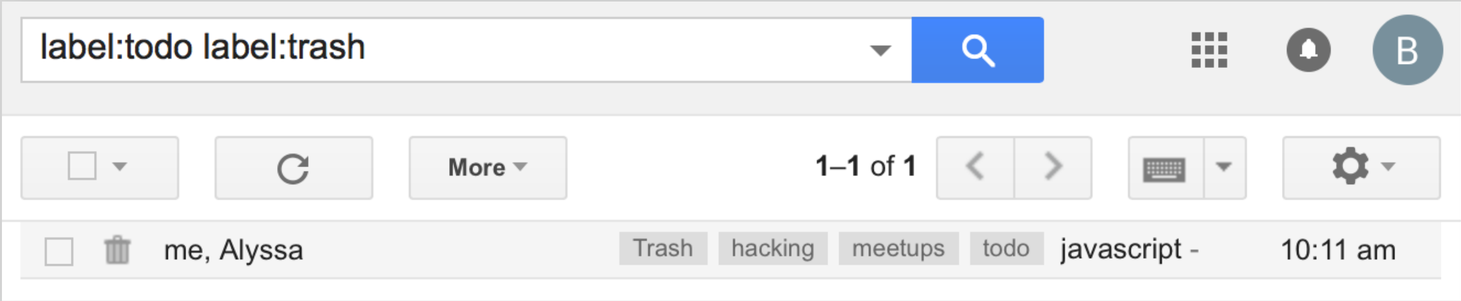
what new problems might this design result in?

a tricky aspect of this synergy

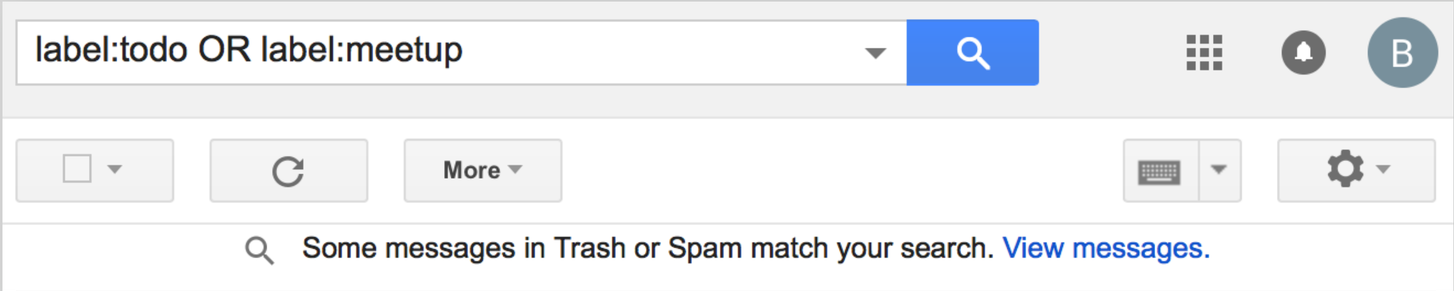
click on trash



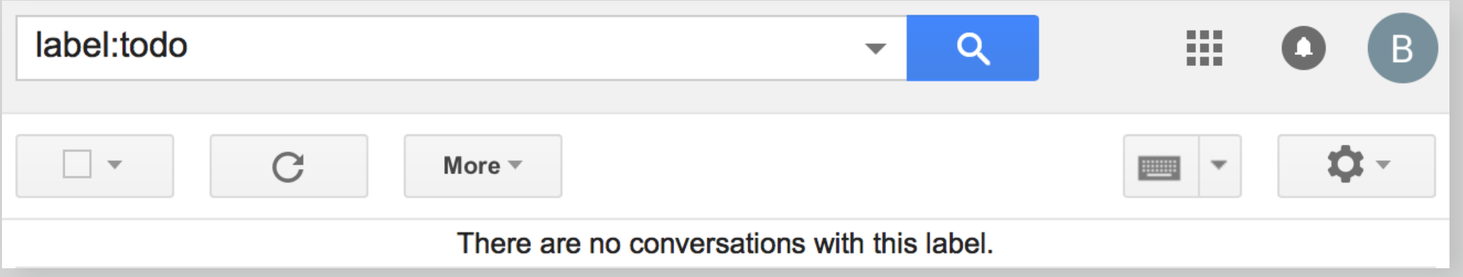
filter on todo and trash



filter on something else



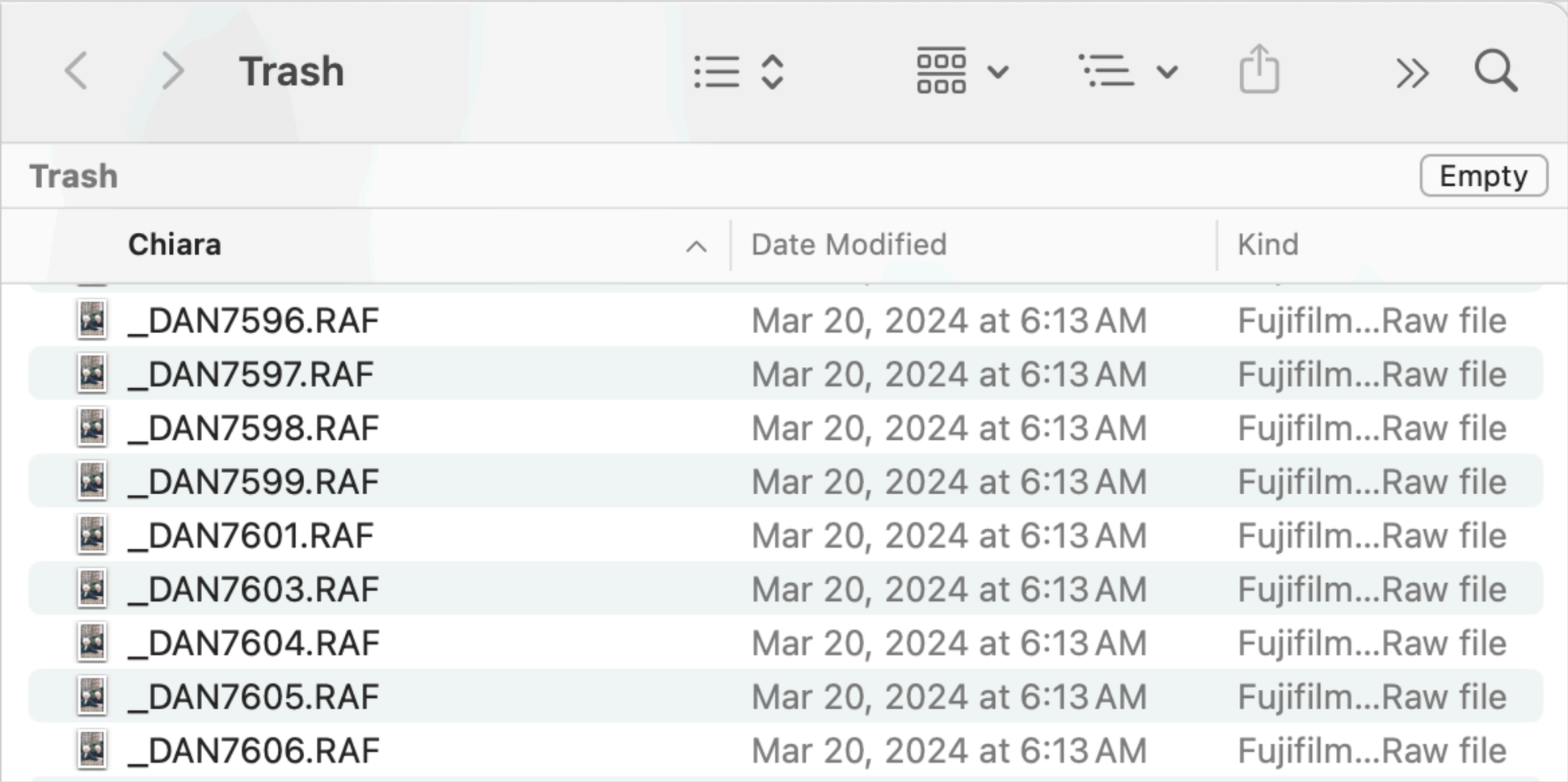
filter on todo label



macOS

trash

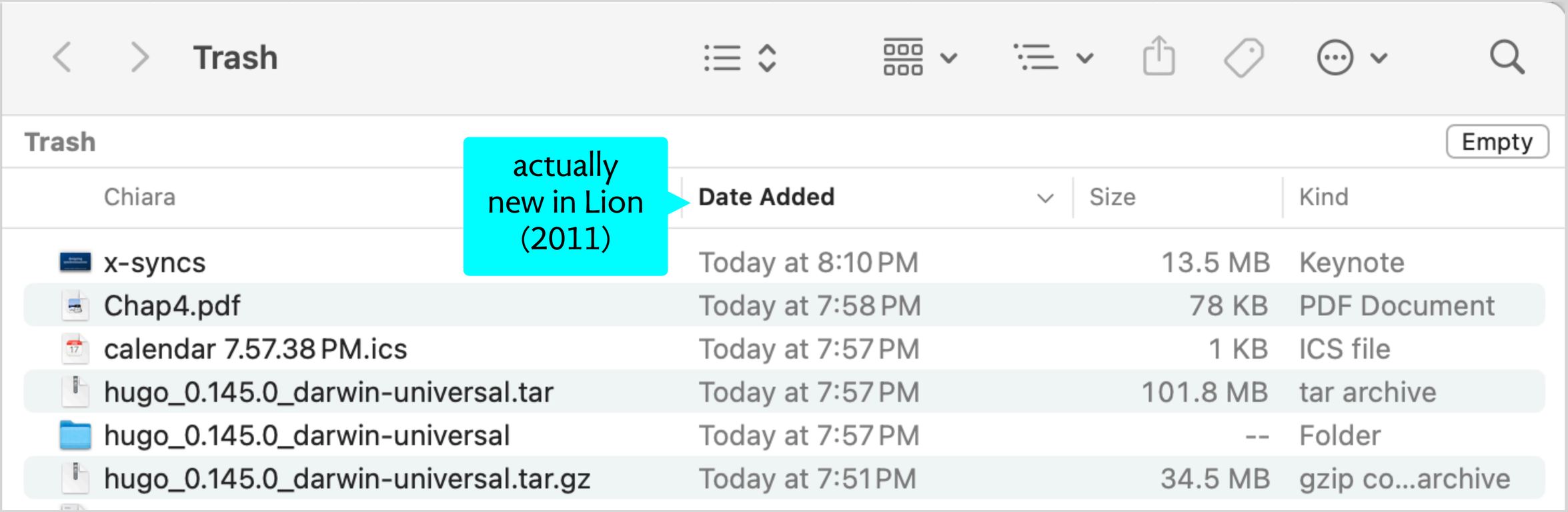
macOS trash is a folder



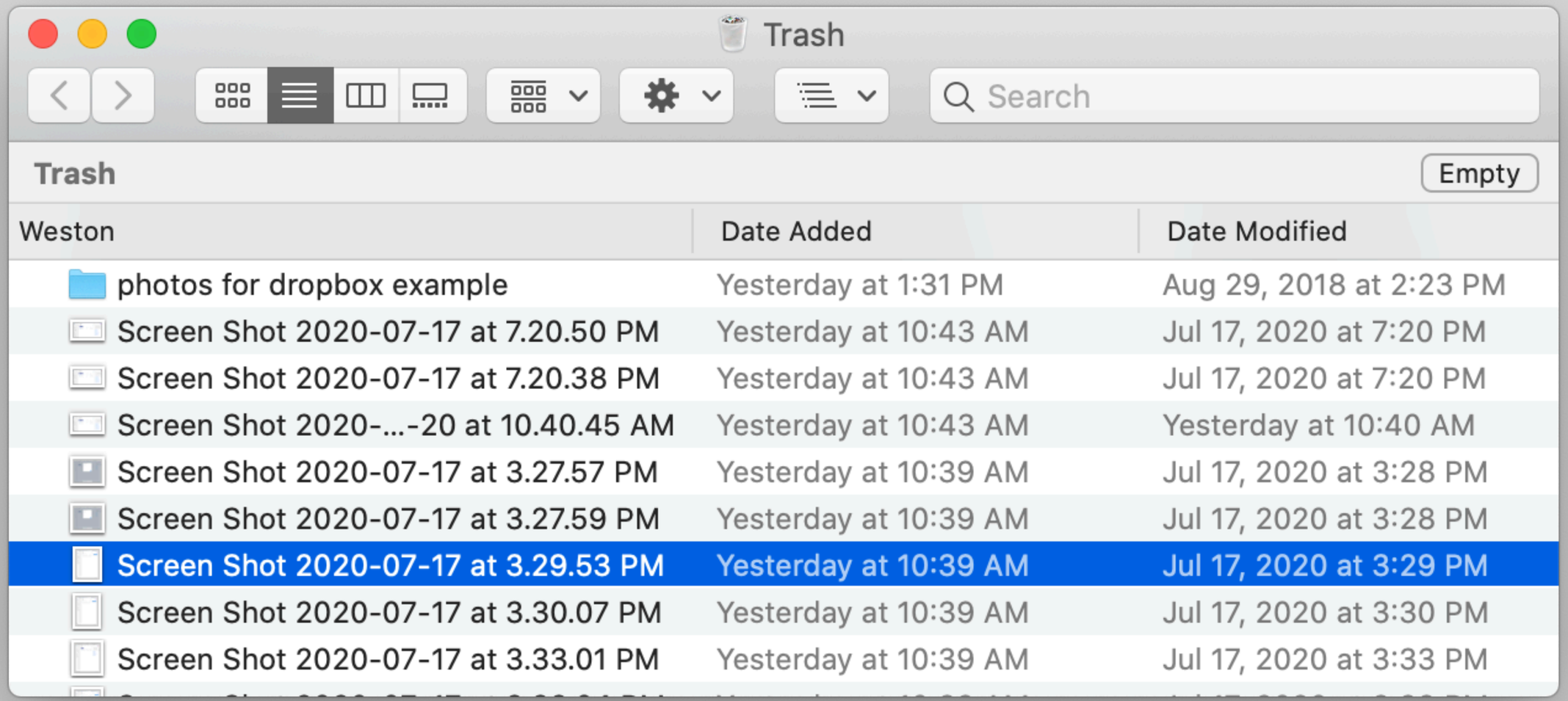
synchronizes Folder and Trash concepts

what benefits does this bring?

how to sort by deletion date?



making the trash a folder



what new problems might this design result in?

hint: macOS has just one trash

Moira
mailing lists

a Moira mailing list

WebMoira List Manager : Daniel Jackson

Help | My Lists | Undo Log (2)

List Name: concept-design
Description: concept design
Attributes: active, moira mailing list
Permissions: private, visible
Last Modified: by dnj with moiraws on 25-nov-2024 19:54:56

Edit

Members

Add Member: Add

Leave List: Remove Me

MIT Users

Daniel Jackson (dnj) remove

Email Addresses

daniel@dnj.photo remove

Administrator

what if we want >1 owner?

Owner: **Daniel Jackson (dnj)**

Change Owner: Change

solution: make the owner a ... mailing list!

WebMoira List Manager : Daniel Jackson

Help | My Lists | Undo Log (1)

List Name: concept-design
Description: concept design
Attributes: active, moira mailing list
Permissions: private, visible
Last Modified: by dnj with moiraws on 25-nov-2024 20:00:17

Edit

what other benefits does this design bring?

Members

Add Member: Add

Leave List: Remove Me

MIT Users

Daniel Jackson (dnj) remove

Email Addresses

daniel@dnj.photo remove

Administrators

Owner: dnj-admin (List)

Change Owner: Change

Add Administrator: Add

Leave Owner List: Remove Me

MIT Users

Daniel Jackson (dnj) remove

remove

moira mailing lists as access control lists

WebMoira List Manager : Daniel Jackson

Help | My Lists | Undo Log (1)

List Name: concept-design
Description: concept design
Attributes: active, moira mailing list
Permissions: private, visible
Last Modified: by dnj with moiraws on 25-nov-2024 20:00:17

Edit

what new problems might this design result in?

Members

Add Member: Add

Leave List: Remove Me

MIT Users

Daniel Jackson (dnj) remove

Email Addresses

daniel@dnj.photo remove

Administrators

Owner: dnj-admin (List)

Change Owner: Change

Add Administrator: Add

Leave Owner List: Remove Me

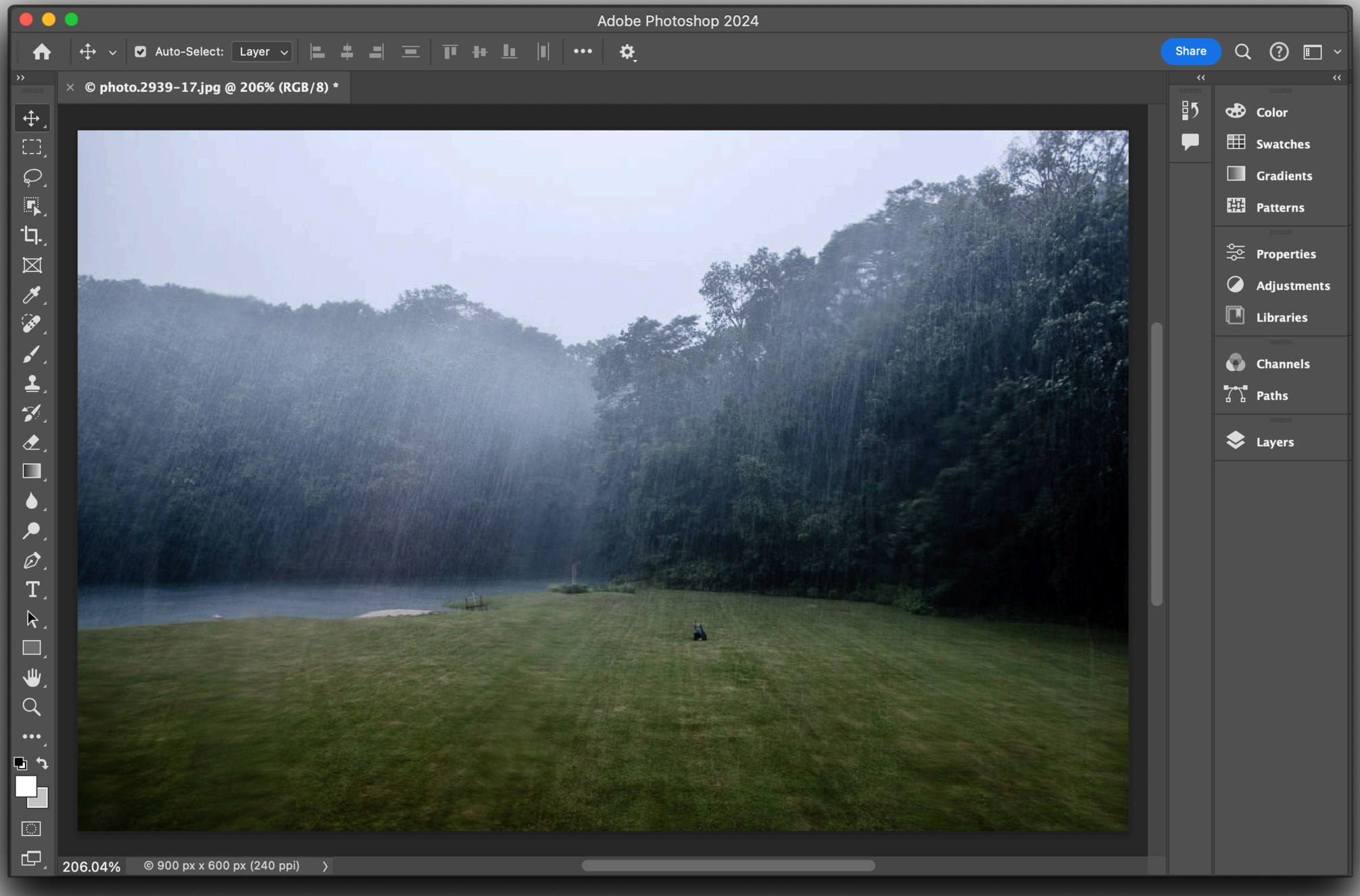
MIT Users

Daniel Jackson (dnj) remove

remove

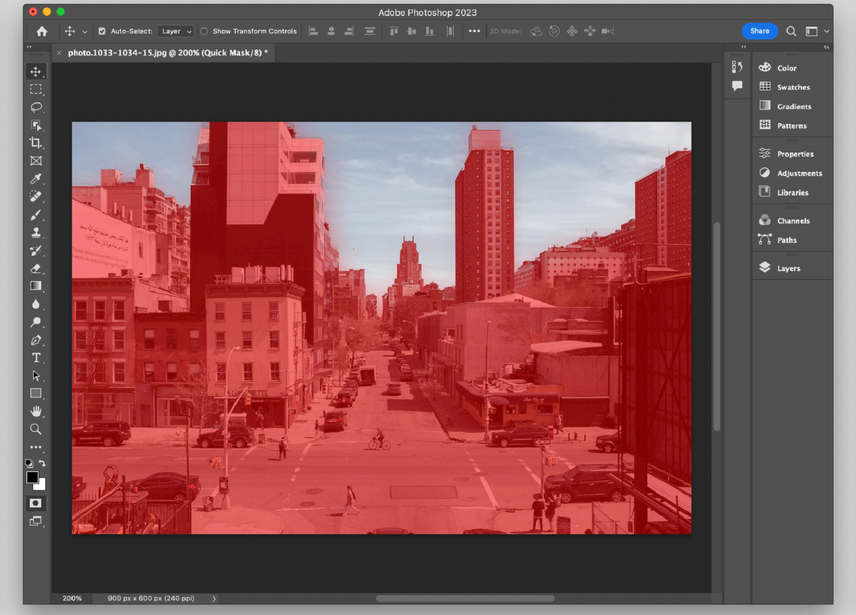
Photoshop selection

how to darken the sky

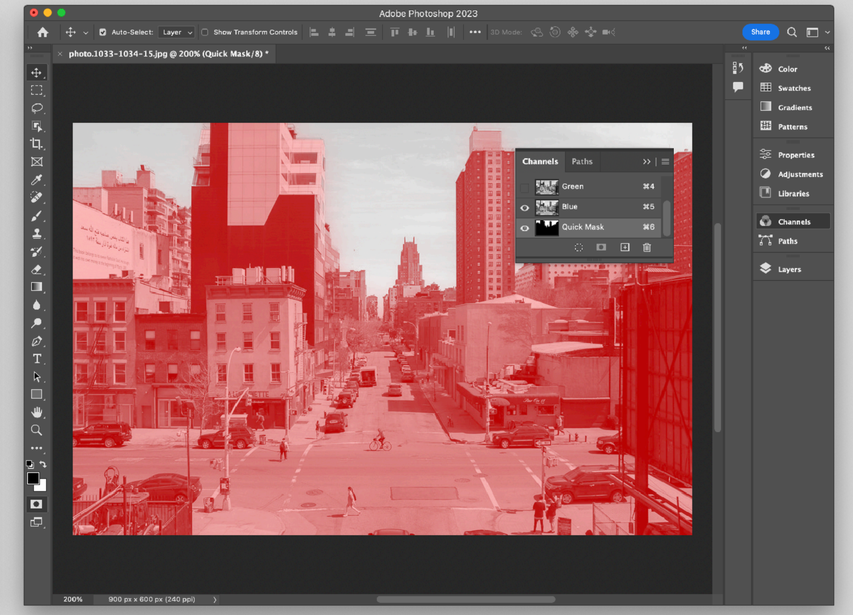


the mother of all synergies

selection (shown as “marching ants”)



edit selection as **mask**

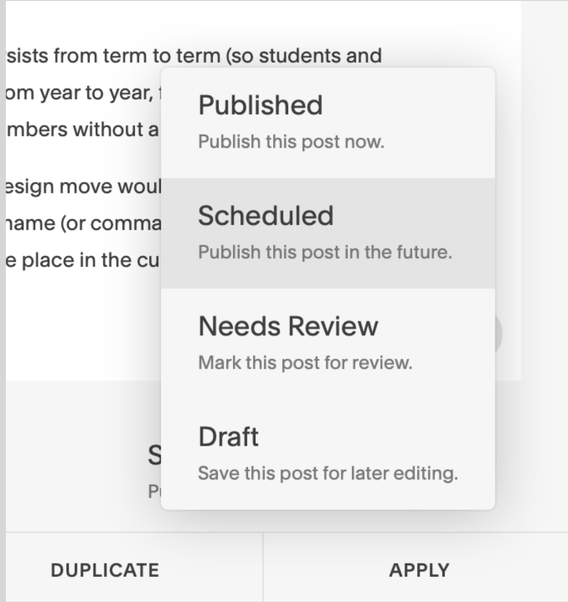


toggle mask as **channel**

selection = mask = channel = grey scale image

Hugo scheduling

a better solution: use the metadata date for scheduling



Squarespace: can schedule blog posts but not other pages, and can change pub date independently (only affects order)

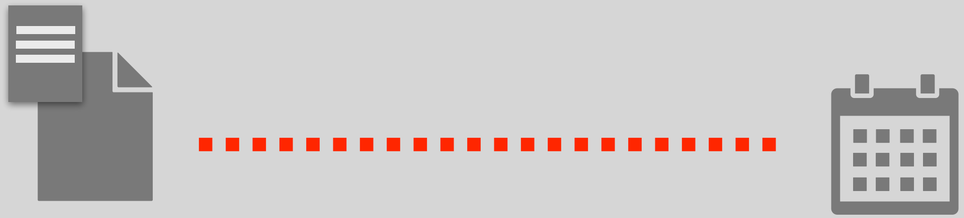
```
---
author: "Daniel Jackson"
title: "The Class Number Dilemma"
date: "2022-02-24"
description: "A Concept Example in Everyday Life"
summary: "Who knew such a simple thing could be so challenging?"

---
```

The Class Number Dilemma

Who knew such a simple thing could be so challenging?

February 24, 2022 · 3 min · Daniel Jackson

Hugo: any page can have date field to schedule, just set date in future



Metadata

Schedule

takeaways

key ideas from this lecture

concept conflation

Zoom: reaction/presence/poll

Spotify: library/playlist, folder/playlist

RealWorld: favoriting/user auth

concept fragmentation

Zoom: presence

RealWorld: favoriting

non-genericity

Spotify: folder

unfamiliarity

Spotify: folder, playlist

concept synergies: powerful but tricky

Gmail: labels/trash

MacOS: folder/trash

what concept design is and isn't

not a magic potion
helps control complexity
not eliminate completely

a framework/language
for structuring designs
exploring collaboratively